Substituting for Real Time and Common Knowledge in Distributed Systems†

Gil Neiger
Sam Toueg

86-790
November 1986
(Revised July 1987)

Department of Computer Science
Cornell University
Ithaca, New York  14853-7501

†Partial support for this work was provided by the National Science Foundation under grant DCR86-01864.
Substituting for Real Time and Common Knowledge in Distributed Systems

Gil Neiger
Sam Toueg
Department of Computer Science
Upson Hall — Cornell University
Ithaca, New York 14853

July 24, 1987

Abstract

We study time and knowledge in synchronous and asynchronous reliable distributed systems. For both types of systems, we describe clocks that can be used as if they were perfectly synchronized real-time clocks in the solution of a large class of problems that we formally characterize. For this same class of problems, we also propose a broadcast primitive that can be used as if it achieves common knowledge. Our clocks and broadcast primitive are tools that considerably simplify the task of designing and proving correct distributed algorithms: the designer can assume that processors have access to real-time clocks and the ability to achieve common knowledge. The latter can be used to implement the abstraction of shared memory.

1 Introduction

Distributed systems with no known bound on relative processor speeds or message transmission times are called asynchronous. Asynchrony makes coordination between processors difficult, and can complicate the design and proof of algorithms for such systems. These difficulties can be reduced if one can assume that all processors have access to perfectly synchronized clocks by which they can coordinate their actions, e.g., that local processor clocks always show real time. Unfortunately, perfect clock synchronization cannot be achieved in asynchronous systems.

In addressing this problem, Lamport defined “logical clocks” [Lamp78]. They were designed to simulate a specific feature of real-time clocks: the ability to capture “potential causality” between events. Lamport argued that in distributed systems this relation between events is often more important than a truly temporal one. He presented an implementation of logical clocks, and used it to solve problems that had earlier seemed to require real-time

*Partial support for this work was provided by the National Science Foundation under grant DCR86-01864
(or perfectly synchronized) clocks. However, no systematic method for determining when logical clocks could substitute for real-time clocks was given. Given an algorithm written assuming that clocks show real time, is the algorithm still correct when logical clocks are used instead?

In attempting to answer this question, we notice that logical clocks as presented by Lamport are lacking in certain respects; for example, an algorithm that terminates when using real-time clocks might not do so when using Lamport’s logical clocks. We correct these problems by modifying Lamport’s clocks. We also formally characterize the class of problems for which solutions using real-time clocks are still correct when these clocks are replaced by our logical clocks. Thus one can solve such problems (and prove the solution correct) with the simplifying assumption that all processors have synchronized real-time clocks, and then run the solution with our logical clocks instead. Note that we characterize a class of problems that can be solved in this way; the particular algorithm used in the solution is irrelevant to us.

Our first goal was to provide a formal substitute for perfectly synchronized real-time clocks. We then use this substitute for real time to circumvent the inability of processors to execute simultaneous actions in asynchronous systems, and hence their inability to achieve “common knowledge”. Thus we are able to use our logical clocks to achieve our second goal: a broadcast primitive that can be used as if it achieves common knowledge.

Processor knowledge and its use in the writing of algorithms have been examined recently by a number of researchers. Of particular interest has been the concept of common knowledge, which is the highest state of knowledge that can be achieved. The ability to achieve common knowledge can be used to implement the abstraction of shared memory, and thus can greatly simplify the design of distributed algorithms. Unfortunately, Halpern and Moses showed that it is impossible to achieve common knowledge of nontrivial facts in asynchronous systems [Halp84]. Just as Lamport proposed a simulation of real-time clocks with his logical clocks, Halpern and Moses suggested modifications to common knowledge that may be achievable. However, they did not present methods to achieve all these variants of common knowledge, or fully characterize the problems and systems for which they can substitute for common knowledge.

To address this question, we describe a primitive, called publication, that achieves timestamped common knowledge [Halp84]. We show that, in solutions to problems in the class we characterized earlier, this primitive can be used as if it achieves common knowledge. Thus, for these problems, our logical clocks and broadcast primitive simplify the task of deriving and proving algorithms: the designer can assume that processors have access to perfectly synchronized real-time clocks and a message passing primitive that achieves common knowledge.

In Section 2 we present our model of distributed systems, as well as a definition of problem specifications, in particular, logical specifications. In Section 3 we discuss local clocks and
describe our implementation of logical clocks. In Section 4 we prove that our logical clocks can substitute for real-time clocks when solving problems with logical specifications. This result is discussed and compared with other work. In Section 5 we extend these results to synchronous systems. In Section 6 we consider notions of knowledge in distributed systems, including timestamped knowledge. In Section 7 we define publications, and in Section 8 we prove that publications can be used as if they achieve common knowledge when solving problems with logical specifications. Section 9 contains a discussion with conclusions.

2 Definitions, Assumptions, and Notation

We first consider asynchronous distributed systems, i.e., systems in which there is no known bound on message transmission times (an extension to synchronous systems is considered in Section 5). We now give a formal model of such a system.

2.1 Distributed Systems

We define a distributed system to be a set $P$ of processors joined by bidirectional communication links. For the sake of simplicity we assume that processors are fully connected.

Each processor has a local memory, part of which is the processor’s state. Let $Q_i$ be the set of possible states of processor $i$. Each processor also has a local clock, which is not part of its state. Clocks are nondecreasing functions of real time, measuring it in discrete units that we call ticks. Thus clock time is a natural number, whereas real time is a non-negative real number.

One processor may send a message to another by placing the message on the link connecting the two processors. The link delivers a message by placing it in a buffer at the other end of the link. A processor receives a message by removing it from this buffer. Information about the buffer is not included in the processor’s state.

2.2 Events

Events are the actions that occur at processors. Let $E_i$ be the set of events executable by processor $i$. All events cause the processor to change its state and to increment its local clock by at least one. There are three types of events. An internal event only causes a change of state. A send event, of the form “send $m$ to $i$”, causes a message with value $m$ to be placed on the link to $i$. A receive event, of the form “receive $m$ from $i$”, causes a message to be removed from the buffer from $i$ (buffer$_i$); this is done only if there is a delivered message in that buffer (otherwise nothing happens). Two special internal events are halt and the null event, $e_\emptyset$. After a processor “halts” it executes no events. Processor $i$ has a transition
function $\delta_i: Q_i \times E_i \rightarrow Q_i$, given the current state, the event executed determines a new state. $\delta_i$ is such that $\forall q \in Q_i, \delta_i(q, e_i) = q$.

2.3 Histories

Associated with a specific execution of a processor is a set of history functions. A clock history function, $\chi_i$, is a function from real time values to clock values i.e., $\chi_i: \mathbb{R} \rightarrow \mathbb{N}$ (let $\mathbb{R}$ be the set of non-negative real numbers). $\chi_i(t)$ is the value of $i$’s local clock at real time $t$. A clock history function, $\chi_i$, is real if $\forall c \in \mathbb{N}[\chi_i(c) = c]$, that is, if $i$’s local clock always shows real time (insofar as it can, as clocks measure time in discrete units). In general, we will use $t, t', t_i$, etc. to refer to real times (in $\mathbb{R}$), and $c, c', c_i$, etc. to refer to clock values (in $\mathbb{N}$).

An event history function, $\eta_i$, is a function from clock values to events, i.e., $\eta_i: \mathbb{N} \rightarrow E_i$. $\eta_i(c) = e$ if processor $i$ began executing $e$ when its local clock showed $c$. Because the execution of an event may take more than one (local) tick, the function $\eta_i$ need not be total. Let $V_i \subseteq \mathbb{N}$ be the set of local clock values on which $\eta_i$ is defined.

Finally, a state history function, $\sigma_i$, is a function from clock values to states, i.e., $\sigma_i: V_i \rightarrow Q_i$. If $\sigma_i(c) = q$, then $i$ is in state $q$ when its local clock reads $c$. We require that after executing event $\eta_i(c)$, processor $i$ is in state $\delta_i(\sigma_i(c), \eta_i(c))$, i.e., that all state transitions are correct.

A global history, $\Gamma$, is a set of clock, event, and state history functions, one for each processor, that corresponds to one execution of the system; i.e., $\Gamma = \{(i, \chi_i, \eta_i, \sigma_i) \mid i \in \mathcal{P}\}$. A global history, $\Gamma$, is a real-time history if $\forall i \in \mathcal{P}[\chi_i$ is real]. In such a history, processor clocks show real time. Note that in a real-time history all local clocks go through every tick.

Consider two global histories $\Gamma_1$ and $\Gamma_2$. If $\eta_{i1} = \eta_{i2}$ and $\sigma_{i1} = \sigma_{i2}$ then in each of these histories the same events and state transitions take place at exactly the same local clock times at processor $i$. Since $i$ cannot observe real time, but only its local clock, it cannot distinguish between $\Gamma_1$ and $\Gamma_2$. We define an equivalence relation, $\sim$, on global histories by

$$\Gamma_1 \sim \Gamma_2 \iff \forall i \in \mathcal{P} [\eta_{i1} = \eta_{i2} \land \sigma_{i1} = \sigma_{i2}].$$

If $\Gamma_1 \sim \Gamma_2$ then no processor can distinguish $\Gamma_1$ from $\Gamma_2$.

A distributed system is identified by the set of global histories that correspond to all executions of the system. We will use $S(\mathsf{messages}, \mathsf{clocks})$ to denote a system where $\mathsf{messages}$ describes the message passing, and $\mathsf{clocks}$ the local clocks. We consider only systems where message passing is reliable, e.g., all messages are eventually delivered some time after they are sent. Asynchronous systems are those in which there is no known bound on message transmission times. $S(\lambda, \lambda)$ denotes asynchronous systems in which the speeds of local clocks are not restricted. $S(\lambda, \mathbb{R})$ denotes asynchronous systems in which local clocks always show real time (and hence are perfectly synchronized). Note that any history in $S(\lambda, \mathbb{R})$ is also in $S(\lambda, \lambda)$, and thus $S(\lambda, \mathbb{R}) \subseteq S(\lambda, \lambda)$. 

4
2.4 Protocols

Processor $i$ runs a local protocol, $\Pi_i$. Given the local clock value and current state of processor $i$, $\Pi_i$ specifies the next event to occur at $i$ ($\Pi_i; \mathbb{N} \times Q_i \rightarrow E_i$). For notational simplicity, we restrict ourselves to deterministic protocols. A sequence of local protocols, $\Pi = (\Pi_i \mid i \in P)$ is called a global protocol, or simply a protocol.

Global history $\Gamma$ is consistent with protocol $\Pi$ if

$$\forall i \in P \forall c \in V_i [\eta_i(c) = \Pi_i(c, \sigma_i(c))]$$

i.e., processors execute exactly those events specified by the protocol. Note that if $\Gamma_1 \sim \Gamma_2$ and $\Gamma_1$ is consistent with protocol $\Pi$ then $\Gamma_2$ is also.

2.5 Problem Specifications

A distributed systems problem can be specified by a predicate on global histories. For example, the serializability problem in distributed databases can be specified by a predicate $\Sigma$ that is satisfied exactly by those histories of the database in which transactions are serializable. In general, a problem specification is a predicate that distinguishes those histories in $S(A, A)$ that solve the problem from those that do not. A protocol $\Pi$ solves a problem with specification $\Sigma$, or satisfies $\Sigma$, if whenever processors run $\Pi$, $\Sigma$ is satisfied. Formally, $\Pi$ satisfies $\Sigma$ in system $S$ if every global history in $S$ consistent with $\Pi$ satisfies $\Sigma$.

A large number of problems in distributed systems have specifications that make no reference to real time. For example, we could specify that transaction execution in a distributed database is serializable without referring to real time. There is no reason to mention real time in the specifications of many problems in distributed systems (e.g., transaction serializability, deadlock prevention and detection, transaction commit, distributed election, eventual agreement, etc.).

We formalize this with what we call logical specifications. A specification $\Sigma$ is logical if for any two equivalent global histories $\Gamma_1 \sim \Gamma_2$, if $\Gamma_1$ satisfies $\Sigma$ then $\Gamma_2$ does also. We note that a specification that does not refer to the clock history functions, $\chi_i$, must be logical and that a specification can refer to real time only by referring to the clock history functions $\chi_i$.

For example, to specify that the local clocks of processors $i$ and $j$ must be synchronized within $\epsilon$ of each other we could write

$$\Sigma \equiv \forall t \in \mathbb{R} \left[ |\chi_i(t) - \chi_j(t)| \leq \epsilon \right].$$

This specification refers to real time and is not logical. We note that a specification may mention real time and still be logical. For example, to specify that every processor eventually executes event $e$ we could write

$$\Sigma \equiv \forall i \in P \exists t \in \mathbb{R} [\eta_i(\chi_i(t)) = e].$$
Even though it refers to real time (using the clock history functions $\chi_i$), this specification is logical; we can write it without referring to real time:

$$\Sigma \equiv \forall i \in P \exists c \in N[\eta_i(c) = e].$$

3 Real-Time and Logical Clocks

3.1 Asynchronous Systems with Real-Time Clocks

In this section we consider asynchronous systems in which processors have real-time clocks and each event takes one (real-time) tick to execute. We will show how to simulate this idealized system in Section 4. Figure 1 models the execution of a protocol $\Pi_i$ in such a system. With each iteration of the while loop the processor executes the event specified by $\Pi_i$ and the local clock (which measures real time) increases by 1. Assuming that message passing is asynchronous, the set of histories corresponding to these executions is $S(\Lambda, R) = \{\Gamma \in S(\Lambda, \Lambda) \mid \Gamma$ is a real-time history $\land$ all events take one tick to execute$\}^1$.

3.2 Asynchronous Systems with Logical Clocks

In practice, hardware clocks cannot exactly measure real time, nor can they be perfectly synchronized. In addressing this problem, Lamport introduced the notion of logical clocks [Lamp78]. Logical clocks are variables (maintained by “software”) that are consistent with the notion of potential causality between events. This notion is formalized as a binary relation, “$\rightarrow$”, on events in a global history: $e_1 \rightarrow e_2$ if one of the following holds:

1. $e_1$ and $e_2$ both occur at the same processor and $e_1$ precedes $e_2$ in real time;

2. $e_1$ is the sending of a message and $e_2$ is the corresponding receive; or

3. $e_2[(e_1 \rightarrow e_2) \land (e_3 \rightarrow e_2)].$

Lamport argued that in distributed systems this “causality” relation between events is often more important than a truly temporal one. He defined logical clocks such that if one event can affect another, then it precedes the other in “logical time”. Formally, histories of systems with logical clocks have the following property:

$$\forall i, j \in P \forall c_1, c_2 \in N[(\eta_i(c_1) \rightarrow \eta_j(c_2)) \Rightarrow (c_1 < c_2)]. \quad (1)$$

---

1This is a slight redefinition of $S(\Lambda, R)$ as given in Section 2.3. The additional assumption that all events take one tick to execute simplifies our notation and strengthens our results since we show how to simulate this more idealized system.
clock = 0;
state = q_0;
while true do
    event = Π_Γ (clock, state); /* protocol determines event to be executed */
    if event = HALT then /* execution of event */
        goto halted;
    else if event = "send m to j" then
        send m to j
    else if event = "receive m from j" then
        if m ∈ buffer_j then
            remove m from buffer_j
        else
            event = ε;
    state = δ_j (state, event); /* corresponding change of state and clock value */
    clock = clock + 1
end;

halted: while true do
    clock = clock + 1

Figure 1: Execution of protocol Π_Γ in a system with real-time clocks

We call (1) the logical clock property. Lamport implemented logical clocks as follows. With the execution of each event a processor’s clock is incremented by 1. All messages are timestamped with the sender’s current clock value. When a processor receives a message it consults its timestamp; the processor’s local clock is then set to one greater than the maximum of the local clock and this timestamp.

In contrast to real-time clocks, Lamport’s logical clocks may stop completely or skip arbitrarily large intervals. This can happen because the values of local clocks depend upon the receipt of messages (or lack thereof). This limits their use as a substitute for real-time clocks. We overcome this limitation by defining logical clocks that never stop and are incremented one tick at a time. These clocks are similar to those of Lamport, and are defined by the implementation given in Figure 2, which illustrates the execution of a protocol Π_Γ using logical clocks. This execution differs from a real-time execution (Figure 1) as follows:
\[\begin{align*}
    \text{clock} &= 0; \\
    \text{state} &= q_0; \\
    \textbf{while true do} & \\
    \text{event} &= \Pi_i(\text{clock, state}); \quad /* \text{protocol determines event to be executed} */ \\
    \text{if} \text{ event} &= \text{HALT} \text{ then} \quad /* \text{execution of event} */ \\
    \text{goto halted;} & \\
    \text{else if} \text{ event} &= \text{"send m to j" then} \\
    &\text{send [clock, m] to j} \quad &\text{else if} \text{ event} &= \text{"receive m from j" then} \\
    &\text{if [clock', m] }\in\text{ buffer}_j \land \text{ clock' < clock then} \\
    &\text{remove [clock', m] from buffer}_j &\text{else} \\
    &\text{event} = e_3 &\text{event} = e_3 \quad &\text{state} = \delta_i(\text{state, event}); \quad /* \text{corresponding change of state and clock value} */ \\
    &\text{clock} = \text{clock} + 1; &\text{clock} = \text{clock} + 1; \\
\end{align*}\]

\textit{halted: while true do}
\[\begin{align*}
    \text{clock} &= \text{clock} + 1
\end{align*}\]

Figure 2: Execution of Protocol \(\Pi_i\) in a system with logical clocks

1. all messages are timestamped with the sender's clock value and are stripped of these timestamps upon arrival (thus the protocol has no access to these timestamps); and

2. a message is received only if its timestamp is less than the local clock value; otherwise it remains in the incoming buffer.

The histories of such systems do not include the timestamps that accompany the messages, only the message bodies. We define \(S(\Lambda, L)\) to be the set of histories of asynchronous systems in which local clocks are our logical clocks.

\textbf{Lemma 1:} Our logical clocks have the logical clock property; that is, if \(\Gamma \in S(\Lambda, L)\) then

\[\forall i, j \in P \forall c_1, c_2 \in \mathbb{N}[(\eta_i(c_1) \rightarrow \eta_j(c_2)) \Rightarrow (c_1 < c_2)].\]
Proof: Left to the reader.

In the sections that follow we refer only to our logical clocks unless we explicitly state otherwise.

3.3 Further Properties of Logical Clocks

Observe that in any \( \Gamma \in S(\mathcal{A}, \mathcal{L}) \) processors never "block"; thus they always continue incrementing their clocks, one tick at a time. Communication is reliable, since messages are still guaranteed to be delivered and available for receipt, and a receiving processor’s clock eventually exceeds any timestamp a message may have. Therefore \( S(\mathcal{A}, \mathcal{L}) \subseteq S(\mathcal{A}, \mathcal{A}) \).

We define a function, \( RT \), that maps any history \( \Gamma \) to the real-time history that \( \Gamma \) would be if the local clocks in \( \Gamma \) were always showing real time. That is, \( \Gamma \) is mapped to history \( RT(\Gamma) \), identical to \( \Gamma \) except that all the clock history functions in \( RT(\Gamma) \) are real (see Section 2.3). Formally, if \( \Gamma = \{ [i, \chi_i, \eta_i, \sigma_i] \mid i \in P \} \), then \( RT(\Gamma) = \{ [i, \chi'_i, \eta_i, \sigma_i] \mid i \in P \} \), where \( \forall i \in P \forall t \in \mathbb{R} \chi'_i(t) = [t] \). Clearly \( \Gamma \sim RT(\Gamma) \) and \( RT(\Gamma) \) is a real-time history.

Note that \( \Gamma \in S(\mathcal{A}, \mathcal{A}) \) does not imply that \( RT(\Gamma) \in S(\mathcal{A}, \mathcal{A}) \). For example, let \( \Gamma \in S(\mathcal{A}, \mathcal{A}) \) be such that processor \( i \) sends \( m \) to \( j \) at time 4 according to \( i \)'s clock, and processor \( j \) receives \( m \) at time 2 to according to \( j \)'s clock. In \( RT(\Gamma) \) all clocks show real time (at discrete intervals), and hence \( i \) sends \( m \) at real time \( t_i, 4 \leq t_i < 5 \), and \( j \) receives \( m \) at real time \( t_j, 2 \leq t_j < 3 \). Then \( t_j < t_i \), so the message is received before it is sent (in real time). This is not possible in the systems we consider (see Section 2.3), and therefore \( RT(\Gamma) \not\in S(\mathcal{A}, \mathcal{A}) \). We can, however, prove the following:

**Theorem 2:** Let \( \Gamma \) be a history in an asynchronous system with logical clocks. Then \( RT(\Gamma) \) is a history in an asynchronous system with real-time clocks. Formally, \( \Gamma \in S(\mathcal{A}, \mathcal{L}) \Rightarrow RT(\Gamma) \in S(\mathcal{A}, \mathcal{R}) \).

Proof: Recall that

\[
S(\mathcal{A}, \mathcal{R}) = \{ \Gamma \in S(\mathcal{A}, \mathcal{A}) \mid \Gamma \text{ is a real-time history \& all events take one tick to execute} \};
\]

we want to show that \( RT(\Gamma) \in S(\mathcal{A}, \mathcal{R}) \). Since \( \Gamma \in S(\mathcal{A}, \mathcal{L}) \), it is clear that in both \( \Gamma \) and \( RT(\Gamma) \) every event takes one tick to execute. Moreover, by definition, \( RT(\Gamma) \) is a real-time history. Thus it suffices to show that \( RT(\Gamma) \in S(\mathcal{A}, \mathcal{A}) \). This will be the case if every send precedes in real time the corresponding receive.

Suppose that for \( \Gamma \in S(\mathcal{A}, \mathcal{L}) \), \( \eta_1(c_1) = e_1 \), \( \eta_2(c_2) = e_2 \), and that \( e_1 \) and \( e_2 \) are corresponding send and receive. In \( RT(\Gamma) \), \( e_1 \) occurs at real time \( t_1 \), \( c_1 \leq t_1 < c_1 + 1 \), and \( e_2 \) at real time \( t_2 \), \( c_2 \leq t_2 < c_2 + 1 \). By Lemma 1, \( c_1 < c_2 \) (and \( c_1 + 1 \leq c_2 \)), so \( t_1 < t_2 \), and \( e_1 \) indeed precedes \( e_2 \) in real time. Thus \( RT(\Gamma) \in S(\mathcal{A}, \mathcal{A}) \), and we are done. 

\[ \square \]
4 Substituting Logical Clocks for Real-Time Clocks

In this section we identify a large class of problems that can be solved using our logical clocks as if they were real-time clocks. In fact, this is the class of problems with logical specifications.

4.1 Proof of Correctness

Suppose a protocol designer derives and proves correct a protocol with the assumption that processors have perfectly synchronized real-time clocks. That is, the programmer proves that real-time histories consistent with this protocol satisfy a given specification, \( \Sigma \). If the protocol is run in a system where processors do not have real-time clocks, it might no longer satisfy \( \Sigma \). We show that if \( \Sigma \) is logical and the processors use our logical clocks, then the protocol is still correct, i.e., it still satisfies \( \Sigma \):

**Theorem 3:** Let \( \Sigma \) be a logical specification. Let \( \Pi \) be a protocol that satisfies \( \Sigma \) when run in an asynchronous system with real-time clocks (i.e., in \( S(\alpha, r) \)). Then \( \Pi \) also satisfies \( \Sigma \) when run in an asynchronous system with our logical clocks (i.e., in \( S(\alpha, L) \)) instead of real-time clocks.

**Proof:** By the assumption on \( \Pi \), any \( \Gamma \in S(\alpha, r) \) that is consistent with \( \Pi \) satisfies \( \Sigma \). We want to show that any \( \Gamma \in S(\alpha, L) \) (the set of histories with our logical clocks) consistent with \( \Pi \) also satisfies \( \Sigma \). Consider such a \( \Gamma \). Since \( RT(\Gamma) \sim \Gamma \), \( RT(\Gamma) \) is also consistent with \( \Pi \) (see the end of Section 2.4). By Theorem 2, \( RT(\Gamma) \in S(\alpha, r) \), so it satisfies \( \Sigma \). Since \( \Sigma \) is logical and \( RT(\Gamma) \sim \Gamma \) then \( \Gamma \) must also satisfy \( \Sigma \). \( \square \)

Suppose we want to derive a protocol that must satisfy some logical specification \( \Sigma \) in an asynchronous system, \( S(\alpha, \lambda) \). To simplify this task, we derive the protocol assuming that the processors' clocks show real time (that is, the underlying system is \( S(\alpha, r) \)). Theorem 3 asserts that running this protocol in an asynchronous system with our logical clocks still satisfies \( \Sigma \).

4.2 An Example

The design of distributed algorithms can be simplified by assuming that processors can use perfectly synchronized real-time clocks to coordinate their actions, and then having them use logical clocks instead. Our example is an algorithm to determine a consistent global state of an asynchronous distributed system. This notion of consistency was defined by Chandy and Lamport [Chan85].

Formally, we define a cut of a global history to be a set of local clock values, one for each processor; e.g., a cut \( C = \{ \epsilon_i \mid i \in P \} \). A cut defines the "prefix" of the global history
up to local time $c_i$ for each processor $i$, and corresponds to a global system state. Cut $C = \{c_i \mid i \in P\}$ is consistent if the following holds:

$$\text{Consistent}(C) \equiv \forall i, j \in P \forall c \in N$$

$$[(c < c_i \land \eta_i(c) = \text{"received } m \text{ from } j\" ) \Rightarrow$$

$$\exists c' \in N[c' < c_j \land \eta_j(c') = \text{"sent } m \text{ to } i\" ]].$$

That is, a consistent cut defines a history prefix in which every message received was sent.

We define an internal event, called store, where a processor saves its local state on stable storage (e.g., a disk). A distributed snapshot protocol is one in which all processors execute store to record states corresponding to a consistent cut. Formally, such a protocol II satisfies the following specification:

$$\Sigma \equiv \exists C[ C = \{c_i \mid i \in P\} \land \text{Consistent}(C) \land \forall i \in P[\eta_i(c_i) = \text{store}]].$$

Our goal is to derive a distributed snapshot protocol, that is, a protocol that satisfies $\Sigma$ in an asynchronous system. To simplify this task we first assume that local clocks show real time. The solution is now immediate: the protocol is simply that at some predetermined real time $t \in N$, all processors execute store. Formally, II is such that $\forall i \in P \forall s_i \in Q_i[\Pi_i(t, s_i) = \text{store}]$.

The proof that II satisfies $\Sigma$ when run in a system with real-time clocks is quite simple. It is clear that each processor executes store. We note that, by definition, the cut defined by the store events can only be inconsistent if it contains the receipt of some message but not its sending. This is impossible since all processors execute store at the same real time; any message received by this time will already have been sent.

Note that $\Sigma$ is logical (there is no reference to the $\chi_i$'s). Thus, by Theorem 3, the protocol is still correct when run with our logical clocks instead of real-time clocks. Our assumption that local clocks are perfectly synchronized simplified the derivation and proof of a distributed snapshot protocol.

Chandy and Lamport also defined the notion of channel states with respect to a consistent cut. With our assumption that clocks are real, it is easy to extend protocol II to also record the channel states.

4.3 Discussion

A large number of protocols have been derived using Lamport's logical clocks in place of real-time clocks (e.g., [Apt84]). They were proven correct using the logical clock property. It has been a "folk theorem" that real-time can be replaced by logical clocks in asynchronous systems. In fact, Morgan derived three specific protocols with the assumption that clocks

---

2 This example also shows why it is essential for logical clocks to move through every tick; otherwise a processor's clock might never reach time $t$ and the processor might never execute store.
are real, and showed that each one of these protocols can use logical clocks instead [Morg85]. For each protocol, Morgan examined how real-time clocks were used in the protocol (and its proof), and then concluded that replacing them with logical clocks would preserve the structure of the specific protocol and its proof.

Morgan's results are immediate from Theorem 3. In fact, Theorem 3 allows us to conclude the correctness of replacing real-time with logical clocks in these protocols, without looking at the protocols or their proofs. We need only note that the specifications of the problems they solve are logical. This is obvious since the statement of each problem does not mention real time.

Therefore Theorem 3 formalizes the folk theorem, and hence simplifies the design of protocols for asynchronous distributed systems.

Theorem 3 holds as long as no bound on message transmission times is assumed in the design and proof of a protocol. It is independent of whether or not message passing is "FIFO"; our method can easily be extended to preserve the order of messages. Furthermore, we can show that Theorem 3 also holds for systems that exhibit the following failure modes: crash failures [Hadz84], send omission failures [Hadz84], general omission failures [Perr86], and arbitrary failures [Lamp82].

We showed how to simulate an idealized system with perfectly synchronized real-time clocks where each event took exactly one real-time tick to execute (i.e., the speeds of processors was identical). However, if when designing an algorithm, one prefers to make different assumptions about the duration of events, then these assumptions can be incorporated into our logical clocks. For example if one designs a protocols assuming that a store event takes at least 20 ticks to executed, then logical clocks should be incremented by 20 after each store event. Our implementation of logical clocks can reflect any such assumptions made about the speed of processors' executions.

Lundelius independently showed that synchronous processors can be simulated by asynchronous ones in a system with asynchronous communication [Lund86]. Her result is extended to systems in which processors may experience different types of failures. She did not, however, formally characterize the class of problems for which her simulation could be applied.

Awerbuch introduces a software device called a "synchronizer" [Awer85]. This is designed to simulate execution in rounds in a totally asynchronous system. Informally, in each round, processors first send messages (according to their states), wait to receive messages sent by other processors in the same round, and then change their states accordingly. In contrast to our approach, Awerbuch's synchronizer is expensive in time and communication complexity: simulating each round of a "round model" protocol requires time and messages in excess of those specified by the simulated protocol itself. The use of logical clocks entails timestamping each message, but no other complexity increase. In particular, no additional messages have to be sent.
5 Synchronous Systems

In the previous sections we showed how to provide the abstraction of perfectly synchronized real-time clocks for asynchronous systems, where clock speeds and message transmission times are unknown. In practice, however, there may be known bounds on both message transmission times and the rates at which processors’ clocks drift with respect to real time. Such systems are called synchronous, and their histories satisfy the following two properties:

S1: all messages arrive within $\Delta$ (real time) after they are sent; and

S2: clocks drift at a rate of at most $\rho$ from real time; that is,

$$\forall i \in P \forall t_1, t_2 \in \mathbb{R}|(1 + \rho)^{-1} \leq \left| \frac{\chi_i(t_1) - \chi_i(t_2)}{t_1 - t_2} \right| \leq (1 + \rho)|.$$

Even in such systems, perfect clock synchronization cannot be implemented. However, in this section we show how to provide the abstraction of perfect clock synchronization for synchronous systems. We do this in two steps: we first provide approximate clock synchronization, and then the abstraction of perfect clock synchronization.

5.1 Providing Approximately Synchronized Clocks

Even though clocks in synchronous systems satisfy S2 above, they may still drift arbitrarily far apart. However, given such a system, it is easy to implement clocks that are approximately synchronized, i.e., that satisfy the following:

S3: clocks are never more than $\epsilon$ apart; that is,

$$\forall i, j \in P \forall t \in \mathbb{R}|\left| \chi_i(t) - \chi_j(t) \right| \leq \epsilon|$$

(such clocks can be implemented even in the presence of arbitrary processor failures [Lund84, Lamp85, Srik85]). Approximately synchronized clocks allow processors some degree of coordination, but not as much as if clocks were perfectly synchronized. In the next section we show how to use approximately synchronized clocks to provide the protocol designer with the abstraction of perfectly synchronized clocks (when solving problems with logical specifications).

5.2 Providing the Abstraction of Perfectly Synchronized Clocks

Let $S(s\Delta, s(\rho, \epsilon))$ denote a synchronous system in which S1, S2, and S3 hold. Then $S(s\Delta, s(0, 0))$ denotes a synchronous system with perfectly synchronized real-time clocks, more simply denoted $S(s\Delta, r)$. Message passing in such a system has the following property:
Lemma 4: Consider a synchronous system with real-time clocks, \( S(s\Delta, r) \). A message sent at local time \( c_s \), with respect to the sending processor, arrives at local time \( c_r \in [c_s, c_s + \Delta] \), with respect to the receiving processor.

Proof: This is immediate because in \( S(s\Delta, r) \) clocks always show real time, and message transmission time is bounded by \( \Delta \).

The correctness of a protocol designed for a synchronous system with perfectly synchronized clocks may rely upon the above property. If the clocks of the underlying system are not perfectly synchronized, this property does not hold. Instead, we note the following:

Lemma 5: Consider a synchronous system with approximately synchronized clocks, \( S(s\Delta, s(\rho, \epsilon)) \). A message sent at local time \( c_s \), with respect to the sending processor, arrives at local time \( c_r \in [c_s - \epsilon, c_s + \Delta'] \), with respect to the receiving processor, where \( \Delta' = \epsilon + (1 + \rho)\cdot\Delta \).

Proof: Let \( \Gamma \in S(s\Delta, s(\rho, \epsilon)) \). Suppose that at real time \( t \) processor \( i \) sends a message to \( j \) and that \( \chi_i(t) = c_s \). By S3, \( \chi_i(t) - \epsilon \leq \chi_j(t) \leq \chi_i(t) + \epsilon \), hence \( c_s - \epsilon \leq \chi_j(t) \leq c_s + \epsilon \). Since \( m \) cannot arrive before real time \( t \), it arrives no earlier than local time \( c_s - \epsilon \), with respect to \( j \).

From the above, \( \chi_j(t) \leq c_s + \epsilon \). By S1, \( m \) arrives no later than real time \( t + \Delta \). By S2, \( \chi_j(t + \Delta) \leq c_s + \epsilon + (1 + \rho)\cdot\Delta = c_s + \Delta' \).

From Lemmas 4 and 5 it is clear that the relation between the local sending and arrival times in \( S(s\Delta, s(\rho, \epsilon)) \) is the same as in \( S(s\Delta', r) \) (where \( \Delta' = \epsilon + (1 + \rho)\cdot\Delta \)), except that an arrival time in \( S(s\Delta, s(\rho, \epsilon)) \) may be before the sending time. Thus, for \( S(s\Delta, s(\rho, \epsilon)) \) to simulate \( S(s\Delta', r) \), it is necessary to delay messages that arrive too “early”, just as in the implementation of our logical clocks. Specifically, we change a normal execution of a protocol running in \( S(s\Delta, s(\rho, \epsilon)) \) as follows:

1. all messages are timestamped with the sender’s clock value and are stripped of these timestamps upon arrival;
2. a message is received only if its timestamp is less than the local clock value; otherwise it remains in the incoming buffer.

We call this delaying of messages logical communication. Let \( S(1\Delta, s(\rho, \epsilon)) \) be the system \( S(s\Delta, s(\rho, \epsilon)) \) with logical communication.

Lemma 6: Consider a synchronous system with approximately synchronized clocks and logical communication, \( S(1\Delta, s(\rho, \epsilon)) \). A message sent at local time \( c_s \), with respect to the sending processor, arrives at local time \( c_r \in [c_s, c_s + \Delta'] \), with respect to the receiving processor, where \( \Delta' = \epsilon + (1 + \rho)\cdot\Delta \).
Proof: Since \( S(t, \Delta, s(\rho, \epsilon)) \subseteq S(s, \Delta, s(\rho, \epsilon)) \) by Lemma 5 we know that \( c_r \in [c_x - \epsilon, c_x + \Delta'] \). The delaying of early messages in \( S(t, \Delta, s(\rho, \epsilon)) \) ensures that the message will not be received before time \( c_x \) (local to the receiver). Thus \( c_r \in [c_x, c_x + \Delta'] \).

From Lemmas 4 and 6 we see that the sending and arrival times in \( S(t, \Delta, s(\rho, \epsilon)) \) have the same relation as in the system with real-time clocks \( S(s, \Delta', r) \), where \( \Delta' = \epsilon + (1 + \rho) \cdot \Delta \). Below we show that \( S(t, \Delta, s(\rho, \epsilon)) \) simulates \( S(s, \Delta', r) \) when solving problems with logical specifications. We first prove the following:

**Theorem 7:** Let \( \Gamma \) be an execution of a synchronous system with approximately synchronized clocks and logical communication. Then \( RT(\Gamma) \) is an execution with perfectly synchronized clocks. Specifically, if \( \Gamma \in S(t, \Delta, s(\rho, \epsilon)) \), then \( RT(\Gamma) \in S(s, \Delta', r) \), where \( \Delta' = \epsilon + (1 + \rho) \cdot \Delta \).

Proof: Let \( \Gamma \in S(t, \Delta, s(\rho, \epsilon)) \); we want to show that \( RT(\Gamma) \in S(s, \Delta', r) \). We first show that in \( RT(\Gamma) \) a message sent at real time \( c_x \) arrives at real time \( c_r \in [c_x, c_x + \Delta'] \).

Since \( \Gamma \in S(t, \Delta, s(\rho, \epsilon)) \), Lemma 6 applies. Hence if in \( \Gamma \) a processor sends a message at local time \( c_x \), it arrives by local time \( c_r \in [c_x, c_x + \Delta'] \). In \( RT(\Gamma) \) local times are real times, so the message is sent at real time \( c_x \) and arrives by real time \( c_r \in [c_x, c_x + \Delta'] \).

By definition, clocks in \( RT(\Gamma) \) always show real time. Thus \( RT(\Gamma) \in S(s, \Delta', r) \).

We now show that protocols designed for synchronous systems with perfectly synchronized real-time clocks can run correctly in systems with approximately synchronized clocks and logical communication.

**Theorem 8:** Let \( \Sigma \) be a logical specification. Let \( II \) be a protocol that satisfies \( \Sigma \) when run in a synchronous system with real-time clocks, \( S(s, \Delta', r) \). Then \( II \) also satisfies \( \Sigma \) when run in a synchronous system with approximately synchronized clocks and logical communication, \( S(t, \Delta, s(\rho, \epsilon)) \), where \( \Delta' = \epsilon + (1 + \rho) \cdot \Delta \).

Proof: By the assumption on \( II \), any \( \Gamma \in S(s, \Delta', r) \) that is consistent with \( II \) satisfies \( \Sigma \). We want to show that any \( \Gamma \in S(t, \Delta, s(\rho, \epsilon)) \) consistent with \( II \) also satisfies \( \Sigma \). Consider such a \( \Gamma \). Since \( RT(\Gamma) \sim \Gamma \), \( RT(\Gamma) \) is also consistent with \( II \) (see the end of Section 2.4). By Theorem 7, \( RT(\Gamma) \in S(s, \Delta', r) \), so it satisfies \( \Sigma \). Since \( \Sigma \) is logical and \( RT(\Gamma) \sim \Gamma \) then \( \Gamma \) must also satisfy \( \Sigma \).

Since we can easily implement approximately synchronized clocks and logical communication in synchronous systems, Theorem 8 states that we can implement the abstraction of perfectly synchronized clocks. Specifically, given a synchronous system, \( S(s, \Delta, s(\rho, \infty)) \), we showed that we can implement a system \( S(t, \Delta, s(\rho, \epsilon)) \) with the following property. To solve a problem with logical specification \( \Sigma \) in \( S(s, \Delta, s(\rho, \infty)) \), we first assume that processes have perfectly synchronized clocks (that is, that the underlying system is \( S(s, \Delta', r) \), where
\( \Delta' = \epsilon + (1 + \rho) \cdot \Delta \). With this simplifying assumption, we derive a protocol and prove it correct. We then run the protocol in \( S(1, \Delta, \Sigma) \). Theorem 8 states that \( \Sigma \) is still satisfied.

### 5.3 Discussion

Given a synchronous system in which clocks drift from each other and from real time, we showed how to provide the simplifying abstraction of perfectly synchronized real-time clocks. This was done by approximately synchronizing the clocks, delaying the arrival of early messages, and making all the remaining clock asynchrony (specified by the parameters \( \rho \) and \( \epsilon \)) manifest itself as a fixed increase in message transmission times.

With this abstraction, when solving a problem with a logical specification, the protocol designer can simply assume that all clocks show real time, and can thus ignore clock parameters \( \rho \) and \( \epsilon \). The resulting protocols and their proofs do not refer to these parameters, and hence are simpler to understand.

Note that the popular "round" model of computation is a special case of a synchronous system with perfectly synchronized clocks, \( S(s, r) \). Thus we can use our abstraction of real-time clocks to automatically run any round model protocol (that solves a problem with a logical specification) on top of any synchronous system that satisfies S1 and S2. This special case was earlier solved by Drummond [Drum86].

### 6 Knowledge in Distributed Systems

The use of knowledge in distributed systems has been intensively studied [Chan86, Fis86, Halp84, Halp85, Mose86a, Nguy86, Pari84, Pari85a, Pari85b]. Several states of knowledge have been identified and some of them were shown to be unattainable in certain distributed systems. In particular, Halpern and Moses showed that common knowledge cannot, in general, be achieved in the asynchronous and synchronous systems described in this paper [Halp84].

Nevertheless, the concept of common knowledge has been crucial in the solution of some important problems in distributed systems [Dwor86, Mose86b].

If processors share memory, the contents of this memory is common knowledge to the processors. Achieving common knowledge of \( \varphi \) is equivalent to "writing" \( \varphi \) in a shared memory. Thus, the ability to achieve common knowledge provides the abstraction of shared memory. This can greatly simplify the derivation of distributed algorithms.

In this section we define timestamped common knowledge, a variant of common knowledge first proposed by Halpern and Moses [Halp84]. In Sections 7 and 8 we develop a primitive that achieves timestamped common knowledge and show that it can be used as if it achieves

\(^3\text{Note that even in our synchronous systems } S(s, S(\rho, \epsilon)) \text{ clocks cannot be perfectly synchronized, and this precludes the attainment of common knowledge.}\)
true common knowledge in solving problems with logical specifications. Thus, this primitive can be used to provide the abstraction of shared memory.

6.1 Defining Knowledge

Let $S$ be a set of histories ($S$ is a given system). A point in a history is a pair $(\Gamma, t) \in S \times R$ that represents the state of the system in history $\Gamma$ at real time $t$. If $\varphi$ is true at this point we say that $(S, \Gamma, t) \models \varphi$. We assume that we have a language for expressing certain ground facts about a system that may or may not be true at certain points (ground facts are those that do not involve the knowledge operators defined below). With each ground fact $\varphi$ we associate the set of points for which the fact is true; call this $\pi(\varphi)$. $(S, \Gamma, t) \models \varphi$ if and only if $(\Gamma, t) \in \pi(\varphi)$.

Halpern and Moses introduced the modality operators $K_i$ to extend a language of ground facts; $K_i \varphi$ denotes that processor $i$ “knows” $\varphi$ [Halp84]. In this paper, we use a modification of their “total view interpretation” of the $K_i$ operators. Processor $i$’s view in $\Gamma$ at time $t$, denoted $v(i, \Gamma, t) = [\chi_i(t), \sigma_i(\chi_i(t))]$ (i.e., $i$’s local clock value and state at time $t$). Informally, $(S, \Gamma, t) \models K_i \varphi$ if and only if $\varphi$ holds at every point in $S \times R$ that $i$ cannot distinguish from $(\Gamma, t)$, that is, if for all $\Gamma', t' \in R$ such that $v(i, \Gamma, t) = v(i, \Gamma', t')$, $(S, \Gamma', t') \models \varphi$. $E \varphi$ (everyone knows $\varphi$) is $\bigwedge_{i \in P} K_i \varphi$. We also define $E^1 \varphi \equiv E \varphi$, $E^{m+1} \varphi \equiv E(E^m \varphi)$, and $(S, \Gamma, t) \models C \varphi$ (common knowledge) if and only if $(S, \Gamma, t) \models E^m \varphi$ for all $m \geq 1$. Halpern and Moses gave a formal fixpoint semantics of common knowledge and also showed that the operators $K_i$, $E$, and $C$ have certain properties, including that they all satisfy the axioms of the logical system S5.

6.2 Achieving Knowledge and Weakenings of Common Knowledge

Halpern and Moses examined how the different states of knowledge described above (and others) might be achieved in distributed systems [Halp84]. They showed that for many systems (including those considered in this paper) it is impossible to achieve $C \varphi$ for most interesting facts $\varphi$. Following this, they proposed weakenings of the notion of common knowledge that could be achieved in distributed systems with various degrees of asynchrony and communication reliability. Among these were $\epsilon$-common knowledge, eventual common knowledge, timestamped common knowledge, and likely common knowledge. Panangaden and Taylor defined concurrent common knowledge, another weakening of common knowledge [Pana86]. We discuss here timestamped common knowledge, as it bears upon our work. As Halpern and Moses gave no formal definition of timestamped knowledge, we do so below.

We introduce a new set of operators, $@_c^i$, for each $i \in P$ and $c \in N$. $@_c^i \varphi$ indicates that $\varphi$ is true when $i$’s clock reads $c$. $(S, \Gamma, t) \models @_c^i \varphi$ if and only if for any $t' \in R$ such that $\chi_i(t') = c$, $(S, \Gamma, t') \models \varphi$. Note that $i$’s clock may or may not have reached $c$ at point $(\Gamma, t)$. 

17
We make the following definitions:

\[ K_{i,c} \varphi \equiv \mathcal{G}_c K_i \varphi, \]
\[ E_c \varphi \equiv \bigwedge_{i \in P} K_{i,c} \varphi, \]
\[ E_c^1 \varphi \equiv E_c, \]
\[ E_c^{m+1} \varphi \equiv E_c(E_c^m \varphi), \]

and \( (S, \Gamma, t) \models C_c \varphi \) (\( \varphi \) is common knowledge timestamped at \( c \)) if and only if \( (S, \Gamma, t) \models E_c^m \varphi \) for all \( m \geq 1 \) (one can show this definition to be consistent with a formal fixpoint semantics).

Halpern and Moses argued that if one achieves timestamped common knowledge of a fact \( \varphi \) in a system in which it is common knowledge that clocks are perfectly synchronized (e.g., \( S(A, \mathcal{R}) \), the set of real-time histories), then one also achieves common knowledge of \( \varphi \) [Halp84]. Using our formal definition of timestamped common knowledge we are able to formalize and prove this. We begin with a technical lemma.

**Lemma 9:** Suppose that \( S \) is a system with perfectly synchronized clocks and that \( (S, \Gamma, t) \models E_c^l \varphi \) (for some \( l \geq 1 \)). Then if \( t_c \in \mathcal{R} \) is such that \( x_i(t_c) = c \) for some \( i \in P \), then \( (S, \Gamma, t_c) \models E_c \varphi \).

**Proof:** By induction on \( l \). We first show the lemma for \( l = 1 \). If \( (S, \Gamma, t) \models E_c \varphi \), then for all \( j \in P \), \( (S, \Gamma, t) \models K_{j,c} \varphi \), that is \( (S, \Gamma, t) \models \mathcal{G}_c K_j \varphi \). Assume \( i \in P \) and \( t_c \in \mathcal{R} \) are such that \( x_i(t_c) = c \). Since clocks are perfectly synchronized, for all \( j \in P \), \( x_j(t_c) = c \). By the definition of \( \mathcal{G}_c \) we have \( (S, \Gamma, t_c) \models K_j \varphi \) for all \( j \in P \). Thus \( (S, \Gamma, t_c) \models E_c \varphi \).

Now assume the lemma holds for values less than or equal to \( l (l \geq 1) \); we show it holds for \( l + 1 \). Assume \( (S, \Gamma, t) \models E_c^{l+1} \varphi \), i.e., \( (S, \Gamma, t) \models E_c(E_c^l \varphi) \). By the base case, if \( t_c \in \mathcal{R} \) is such that \( x_i(t_c) = c \) for some \( i \in P \) then \( (S, \Gamma, t_c) \models E(E_c^l \varphi) \). Consider \( \Gamma' \in S \) and \( t' \in \mathcal{R} \) such that \( v(j, \Gamma, t_c) = v(j, \Gamma', t') \) for any \( j \in P \); by the definition of \( E_c \), \( (S, \Gamma', t') \models E_c \varphi \). Since clocks are perfectly synchronized, \( \Gamma' \) is such that \( x_j(t_c) = x_i(t_c) = c \), and since \( v(j, \Gamma, t_c) = v(j, \Gamma', t_c), \)
\( x_j(t') = x_i(t_c) = c \). So, by the induction hypothesis (with \( \Gamma', t', \) and \( j \)), \( (S, \Gamma', t') \models E_c \varphi \). Thus \( (S, \Gamma, t_c) \models K_j(E_c \varphi) \), and since \( j \) was chosen arbitrarily, \( (S, \Gamma, t_c) \models E_c(E_c \varphi) \). Thus \( (S, \Gamma, t_c) \models E_c^{l+1} \varphi \) and we are done.

**Theorem 10:** Suppose that \( S \) is a system with perfectly synchronized clocks and that \( (S, \Gamma, t) \models C_c \varphi \). Then if \( t_c \in \mathcal{R} \) is such that \( x_i(t_c) = c \) for some \( i \in P \), then \( (S, \Gamma, t_c) \models C_c \varphi \).

That is, if the system achieves \( C_c \varphi \) then \( \varphi \) will be (or was) common knowledge at the real time when clocks show (or showed) \( c \).
Proof: \((S, \Gamma, t) \models C_\varphi\), so by definition \((S, \Gamma, t) \models E^l_\varphi\) for all \(l \geq 1\). By Lemma 9, since \(t_c\) is such that \(\chi_i(t_c) = c\) then \((S, \Gamma, t_c) \models E^l_\varphi\) for all \(l \geq 1\). Thus by definition \((S, \Gamma, t_c) \models C_\varphi\). □

We will later use Theorem 10 to show that timestamped common knowledge can substitute for common knowledge in the solutions of problems with logical specifications.

### 6.3 Knowledge-Based Protocols

Halpern and Fagin introduced "knowledge-based protocols" for distributed systems [Halp85]. We consider a simplification of their work and adapt it to our definitions.

In Section 2 we defined protocols to be functions mapping processor states and local clock values to events. These are what Halpern and Fagin called "simple protocols." A knowledge-based protocol also uses a processor's knowledge (as defined in Section 6.1) in determining the events to be executed. A protocol designer may use the knowledge operators \(K_i\), \(E\), \(E^m\), and \(C\) when writing such protocols.

A processor's knowledge is the set of facts that it "knows" according to the definitions of Section 6.1: it is a function of the processor's current view and the set \(S\) of global histories that it believes are possible. \(S\) is called the admissible set. Let \(F\) be the set of all well-formed formulas composed of ground facts, logical connectives, and knowledge operators. The set of formulas that processor \(i\) knows at point \((\Gamma, t)\), using \(S\) as admissible set, is given by the function \(F_i\):

\[
F_i(v(i, \Gamma, t), S) = \{ \varphi \in F \mid (S, \Gamma, t) \models K_i \varphi \}.
\]

Given the current clock value, state, and knowledge of processor \(i\), a knowledge-based protocol \(\Pi_i\) specifies the next event to be executed at \(i\): \(\Pi_i \colon N \times Q \times 2^F \rightarrow E_i\). That is, if a processor is in state \(s\) at local time \(c\), and it is using admissible set \(S\), then it knows the facts in \(F_i([c, s], S)\), and should execute \(\Pi_i(c, s, F_i([c, s], S))\). We say that global history \(\Gamma\) is consistent with knowledge-based protocol \(\Pi\) and admissible set \(S\) if

\[
\forall i \in P \forall c \in V_i[\eta_i(c) = \Pi_i(c, \sigma_i(c), F_i([c, \sigma_i(c)], S))];
\]

that is, if all processors always execute the events specified by protocol \(\Pi_i\).

Usually, the admissible set \(S\) will be exactly the set of histories that corresponds to the system being run. We will, however, have occasion to use sets that correspond to other systems. The idea of defining knowledge with a set of histories other than those of the system being run was also considered by Fischer and Immerman [Fisc86].

### 7 Publications

In this section we first define a message passing primitive, denoted "publication", which can be used in knowledge-based protocols as if it achieves common knowledge. To use this
primitive a processor executes the event “publish m”. A publication can be thought of as a broadcast (a message sent to all processors) that arrives everywhere at the same local time; that is, there will be some time c (the publication time) such that each processor receives the broadcast when its local clock reads c. The broadcast of the message becomes timestamped common knowledge with c as timestamp.

We then describe an implementation of publications for asynchronous systems (with logical clocks) and one for synchronous systems (with approximately synchronized clocks and logical communication). With each implementation, Theorems 3 and 8, respectively, state that publication times are “equivalent” to real times (publishing m with publication time c then becomes equivalent to writing m in a shared memory at real time c). Theorem 10 is then used to show that our implementations of publications can be used as if they achieve common knowledge. This will be proven formally in Section 8.

7.1 Defining Publications

The execution of a knowledge-based protocol with publications is modeled by Figure 3 (this is an execution of an asynchronous system with logical clocks). The differences between this execution and those described in Section 3 are the following. Pi is a knowledge-based protocol, so the next event to be executed is a function not only of clock and state, but also of the processor’s knowledge as computed by the function Fi. Each processor can now publish messages by executing “publish” events. Processor i keeps a publication table, denoted PTi (see Figure 4), to maintain information about published messages. Each entry in PTi corresponds to a “publish” event, and has four fields:

1. the name of the processor that initiated that publication;
2. the value of the local clock of the publishing processor (at the time the publication was initiated);
3. the message to be published;
4. the time at which the message is published.

(1) and (2) serve to uniquely identify the publication. (We will see that these individual publication tables simulate one shared table.)

The definition of global histories (Section 2.3) is extended to include publications. The event sets, Ei, include publish events. A global history, \( \Gamma = \{[i, \chi_i, \eta_i, \sigma_i, \tau_i] \mid i \in P\} \), now includes a publication table history function, \( \tau_i \), for each processor i; \( \tau_i(c) = PT_i \) if when i’s clock reads c, its publication table is PTi. Processor i’s view at time t includes the value of i’s publication table at time t; that is, \( v(i, \Gamma, t) = [c, \sigma_i(c), \tau_i(c)] \), where \( c = \chi_i(t) \). \( \Gamma_1 \sim \Gamma_2 \) if and only if

\[
\forall i \in P[\eta_{1i} = \eta_{2i} \land \sigma_{1i} = \sigma_{2i} \land \tau_{1i} = \tau_{2i}].
\]
\[\text{clock} = 0;\]
\[\text{state} = q_0;\]
\textbf{while true do}
\[\text{view} = \{\text{clock, state, } PT\};\]
\[\text{knowledge} = F(t)_{\text{view}, S};\]
\[\text{event} = \Pi_i\{\text{clock, state, knowledge}\};\]
\text{if event = \text{HALT} then}
\[\text{goto halted};\]
\text{else if event = "send m to j" then}
\[\text{send } [\text{clock, m}] \text{ to } j;\]
\text{else if event = "receive m from j" then}
\[\text{if } [\text{clock}', m] \in \text{buffer}_j \wedge \text{clock}' < \text{clock} \text{ then}
\[\text{remove } [\text{clock}', m] \text{ from buffer}_j;\]
\text{else}
\[\text{event} = e_0;\]
\text{else if event = "publish m" then}
\[\text{Publish}(m);\]
\[\text{state} = \delta_i(\text{state, event});\]
\[\text{clock} = \text{clock} + 1;\]
\textbf{end;}

\textbf{halted: while true do}
\[\text{clock} = \text{clock} + 1\]

\text{Figure 3: Execution of knowledge-based protocol } \Pi_i \text{ with publications and logical clocks, using } S \text{ as admissible set}
<table>
<thead>
<tr>
<th>Initiating Processor</th>
<th>Initiation Time</th>
<th>Message Body</th>
<th>Publication Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>&quot;z = 2&quot;</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>&quot;y = 1&quot;</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4: Example of a publication table

Informally, \( \Gamma_1 \sim \Gamma_2 \) if in both histories each processor executes the same events from the same states and publication tables, at the same local times. Processors cannot distinguish two equivalent histories \( \Gamma_1 \sim \Gamma_2 \) without access to real time. We note that if \( \Gamma_1 \sim \Gamma_2 \) and \( \Gamma_1 \) is consistent with \( \Pi \) and admissible set \( S \), then \( \Gamma_2 \) is also.

Executions with publications must satisfy two publication axioms, formally defined below:

P1: \([i, c_i, m, c] \in \tau_j(c_j) \Rightarrow c_i < c \land \eta_i(c_i) = \text{"publish } m\text{"}];

P2: \(\eta_i(c_i) = \text{"publish } m\text{"} \Rightarrow \exists c \in \mathbb{N}[c_i < c \land \forall j \in P[[i, c_i, m, c] \in \tau_j(c)]].\)

(In the above all free variables are assumed to be universally quantified.) P1 asserts that any entry in any processor's publication table corresponds to an executed publication. P2 asserts that for every publication executed, there is a time, the publication time, such that a corresponding (identical) entry exists in the publication table of each processor at that local time. This entry includes the publication time, as well as the other fields noted above.

Let \( S_P \) be a system with certain message passing and clock characteristics. We denote by \( S_P \) the system with the same characteristics and publication events (the histories of \( S_P \) satisfy P1 and P2).

### 7.2 Using Publications to Achieve Timestamped Common Knowledge

We show that publications achieve timestamped common knowledge. Consider a system \( S_P \) with publications. We begin with a technical lemma.

**Lemma 11:** If \((S_P, \Gamma, t_c) \models [i, c_i, m, c] \in PT_j \) for some \( j \) such that \( \chi_j(t_c) = c \) then for all \( l \geq 1 \) \((S_P, \Gamma, t_c) \models E_i^l \varphi \), where \( \varphi \equiv [\eta_i(c_i) = \text{"publish } m\text{"}] \).

**Proof:** We first note that by P1, \((S_P, \Gamma, t_c) \models \varphi \). The proof is now by induction on \( l \).

To prove the base case \((l = 1) \), we must show that \((S_P, \Gamma, t_c) \models E_i \varphi \), i.e., for all \( k \in P \), \((S_P, \Gamma, t_c) \models K_{k \cdot c} \varphi \). Consider such a \( k \in P \), and let \( t \in R \) such that \( \chi_k(t) = c \). By P2
$(S_P, \Gamma, t) \models [i, c_i, m, c] \in PT_k$. Let \( \Gamma' \subseteq S_P \) and \( t' \in R \) be such that \( v(k, \Gamma', t') = v(k, \Gamma, t) \). Then \((S_P, \Gamma', t') \models [i, c_i, m, c] \in PT_k\), and by P1 \((S_P, \Gamma', t') \models \varphi\). Thus \((S_P, \Gamma, t) \models K_k \varphi\), so by definition \((S_P, \Gamma, t_c) \models K_{k,c} \varphi\) and we are done.

Now assume the lemma holds for \( t \); we must show that \((S_P, \Gamma, t_c) \models E^{i+1}_c \varphi\). It suffices to show that for all \( k \in P \) \((S_P, \Gamma, t_c) \models K_{k,c}(E^i_c \varphi)\), i.e., \((S_P, \Gamma, t_c) \models E^i_k K_k(E^i_c \varphi)\). Let \( t \) be such that \( \chi(t) = c \). By P2 \((S_P, \Gamma, t) \models [i, c_i, m, c] \in PT_k\). Let \( \Gamma' \subseteq S_P \) and \( t' \in R \) be such that \( v(k, \Gamma', t') = v(k, \Gamma, t) \). Note that \( \chi_k(t') = \chi_k(t) = c \) and \((S_P, \Gamma', t') \models [i, c_i, m, c] \in PT_k\). Then by the induction hypothesis (with \( \Gamma', t', \) and \( k \)) , \((S_P, \Gamma', t') \models E^i_c \varphi\). Thus \((S_P, \Gamma, t) \models K_k(E^i_c \varphi)\), so \((S_P, \Gamma, t_c) \models E^i_k K_k(E^i_c \varphi)\), and we are done. \[ \square \]

We now prove that if processor \( i \) executes "publish \( m \)" at local time \( c_i \), then it will become timestamped common knowledge (at the publication time) that \( i \) published \( m \) at local time \( c_i \).

**Theorem 12:** Let \( \varphi \equiv [\eta_i(c_i) = \text{"publish m"}] \). If \((S_P, \Gamma, t) \models \varphi\), then there are \( c \in N \) and \( t_c \in R \) such that \((S_P, \Gamma, t_c) \models C_c \varphi\).

**Proof:** Since \((S_P, \Gamma, t) \models \varphi\), from P2, there is a \( c > c_i \) where for every \( j \in P \) \([i, c_i, m, c] \in \tau_j(c).\)

Let \( t_c \in R \) be such that \( \chi_j(t_c) = c \) for some \( j \in P \); then \((S_P, \Gamma, t_c) \models [i, c_i, m, c] \in PT_j\). By Lemma 11, \((S_P, \Gamma, t_c) \models E^l_c \varphi\) for all \( l \geq 1 \); therefore \((S_P, \Gamma, t_c) \models C_c \varphi\). \[ \square \]

Although we have only shown that it is the execution of the publication, \( \varphi \), that becomes timestamped common knowledge, we could also show that other facts relevant to the publication (e.g., the contents of the message published) also become timestamped common knowledge. Furthermore, once the fact \( \varphi \) becomes true, it remains so. Hence, we could also show that \( \varphi \) will be timestamped common knowledge for any timestamp greater than or equal to the publication time.

A corollary to Theorems 10 and 12 shows that in systems with real-time clocks, publications achieve true common knowledge.

**Corollary 13:** Let \( S_P \) be a system with publications and real-time clocks and let \( \varphi \equiv [\eta_i(c_i) = \text{"publish m"}] \). If \((S_P, \Gamma, t) \models \varphi\), then there is a \( c \in N \) such that \((S_P, \Gamma, c) \models C \varphi\).

**Proof:** By Theorem 12 there is a \( c \in N \) and \( t_c \in R \) where \((S_P, \Gamma, t_c) \models C \varphi\). Since \( S_P \) contains only real-time histories, \( \chi_i(c) = c \) for all \( i \in P \). Clocks in \( S_P \) are perfectly synchronized, and hence, by Theorem 10, \((S_P, \Gamma, c) \models C \varphi\). \[ \square \]

Corollary 13 states that if a message is published in a system with real-time clocks then the publication becomes common knowledge. From this, one can show that publications cannot be implemented in asynchronous systems with real-time clocks (i.e., in \( S(\lambda, r) \)), as this would contradict the impossibility results of Halpern and Moses [Halp84].
procedure Publish(m);

    send [PUBLISH, clock, m] to all j \in P;
    for each j \in P do in parallel
        await [PROPOSED_TIME, t_j] from j;
        maxtime = max\{t_j | j \in P\};
    send [PUBLISHED, maxtime] to all j \in P;

end Publish;

Figure 5: The procedure Publish

7.3 Publication in Asynchronous Systems

In this section we give an implementation of publications for asynchronous systems with logical clocks, S(A, t). In Section 8.1, we show that this implementation provides the abstraction of achieving common knowledge. Some of the ideas used in the implementation are from a protocol due to Skeen that was used in the ABCAST primitive of Birman and Joseph [Birm87].

7.3.1 An Implementation

With the exception of the procedure Publish the implementation was given in Figure 3. The idea behind the implementation is as follows. The publishing processor sends the message to all processors (including itself), and requests from each a proposed publication time. Each processor responds with its local clock value (incremented by 1) and waits until it hears again from i. Processor i takes the maximum of all proposed times (this will be the agreed publication time), and sends it to all processors, who then make the appropriate entries in their publication tables.

Procedure Publish is given in Figure 5. Note that Publish uses two new programming constructs: await and do in parallel. To execute an await, the processor stops and waits for the indicated message. The do in parallel construct is self-explanatory. In the procedure Publish a processor may be awaiting messages from several processors at the same time; it processes each as it arrives.

When a processor receives a message tagged publish (even from itself) it executes the
procedure ProcessPubl;
    receive [PUBLISH, init, m] from i;
    send [PROPOSEDTIME, clock + 1] to i;
    await [PUBLISHED, maztime] from i;
    put [i, init, m, maztime] in publication table;
end ProcessPubl;

Figure 6: The procedure ProcessPubl

procedure ProcessPubl (Figure 6). ProcessPubl is executed immediately upon the receipt of a PUBLISH message. Several invocations of ProcessPubl may run in parallel with each other, and with an execution of Publish. However, all other execution is suspended until after all executions of ProcessPubl complete; in particular, local clocks do not advance.

Define $S_P(\lambda, L)$ to be the set of histories corresponding to executions of this implementation in asynchronous systems with our logical clocks.

7.3.2 Proof of Correctness

We now prove the correctness of this implementation of publications:

Theorem 14: Consider a global history $\Gamma$ in an asynchronous system with logical clocks and our implementation of publications, i.e., $\Gamma \in S_P(\lambda, L)$. Then $\Gamma$ satisfies publication axioms P1 and P2, and $\Gamma \in S_P(\lambda, \lambda)$. Thus $S_P(\lambda, L) \subseteq S_P(\lambda, \lambda)$.

Proof: We first show that the clocks in $\Gamma$ are nondecreasing functions of real time. We have already seen (in Section 3.3) that with the exception of publishing, this will be true of executions of Figure 3. Any executions of Publish and ProcessPubl will terminate after a finite amount of time (all messages are guaranteed to eventually arrive, and are processed as soon as they do), so a processor may never get “trapped” in one iteration of the while loop of Figure 3. Thus the processor will eventually reach the bottom of the while loop and increment its clock. (We do not consider here the possibility of “livelock”. This could potentially occur if a few processors were very fast and flooded the system with PUBLISH messages, forcing the other processors to only execute ProcessPubl and never advance to the end of their while loops. We can modify our implementation to avoid such “livelock”.)
We now show that our implementation satisfies P1 and P2.

P1 states that if at local time $c_t$ processor $j$ has $[i, c_t, m, c]$ in its publication table, then it must have executed $ProcessPubl$ and received $[Publish, c_t, m]$, and later $[Published, c]$, from $i$. Since processor $i$ sent $[Publish, c_t, m]$, it must have executed $Publish$ when its local clock read $c_t$. By the way $Publish$ and $ProcessPubl$ work, $c$ must be the $maztime$ value as established by $i$, and $c \geq c_t + 1$ (or $c_t < c$).

P2 states that if processor $i$ executes “publish $m$” at local time $c_i$, then there will be some $c, c > c_i$, such that a corresponding entry exists in the publication time of every processor at local time $c$. Suppose $i$ executes the publish as indicated. Through the executions of $Publish$ and $ProcessPubl$, it determines a value, $maztime$, that is in advance of the clocks of all processors (specifically, $maztime > c_i$). $i$ will thus have $[i, c_i, m, maztime]$ in its publication table enter $[i, c_i, m, maztime]$ The same will be true for all other processors, since they will all enter $[i, c_i, m, maztime]$ in their publication tables before they increment their local clocks. That by the time the local clock of a processor reads $maztime$, it will have $[i, c_i, m, maztime]$ in its publication table. Thus P2 is satisfied, using $maztime$ for $c$.

Axioms P1 and P2 are satisfied. It should be clear that normal communication in $\Gamma$ is still reliable. Thus $\Gamma \in S_P(\lambda, \lambda)$.

### 7.4 Publications in Synchronous Systems

In the previous sections, we described an implementation of publications for asynchronous systems. This implementation required three phases of communication and that processors block between these phases. If the underlying system is synchronous, we can provide a simpler and more efficient (one phase, non-blocking) implementation of publications.

Given a synchronous system that satisfies S1 and S2 (see Section 5 above) for some $\Delta$ and $\rho$ we first synchronize clocks approximately; this results in a system $S(s\Delta, s(\rho, \epsilon))$ for some $\epsilon$. We then “delay” early messages as explained in Section 5.2. This results in the system $S(l\Delta, s(\rho, \epsilon))$. By Lemma 6, a message sent in such a system at local time $c$ is guaranteed to arrive by local time $c + \Delta'$, where $\Delta' = c + (1 + \rho)\Delta$.

We now describe a one-phase implementation of publications for $S(l\Delta, s(\rho, \epsilon))$. To publish message $m$, processor $i$ sends the message $[Publish, c, m]$, where $c$ is the value of $i$’s clock at the time this message is sent. When this message arrives at processor $j$, the entry $[i, c, m, c + \Delta']$ is made in $j$’s publication table, i.e., $m$ is published at local time $c + \Delta'$. Since $m$ is guaranteed to arrive by $c + \Delta'$ (local time), we can easily show that this implementation satisfies the publication axioms. We call this system with publications $S_P(l\Delta, s(\rho, \epsilon))$.

Let $S_P(s\Delta, \rho)$ be a synchronous system with publications and perfectly synchronized real-time clocks. By Corollary 13 we know that in this system publications achieve common knowledge. In Section 8.3 we will show that our implementation of publications for syn-
chronous systems, \(S_P(l\Delta, s(\rho, e))\), simulates \(S_P(s\Delta, r)\), and hence provides the abstraction of achieving common knowledge (and of real-time clocks).

8 Substituting for Common Knowledge

We show how our implementations of publications can be used to substitute for common knowledge in both asynchronous and synchronous systems. When solving problems with logical specifications, programmers can use these implementations as if they achieve common knowledge. Thus in many cases the impossibility of achieving common knowledge can be effectively circumvented.

8.1 Providing the Abstraction of Common Knowledge in Asynchronous Systems

Consider the design of a knowledge-based protocol for a problem with a logical specification in an asynchronous system. We have already shown that the designer can assume that local time (as provided by our logical clocks) is real time (Theorem 3). We also showed that our implementation of publications “publishes” messages at the same local time at all processors. If this local time is real time, then, by Corollary 13, publications achieve common knowledge. If the designer uses our logical clocks, then he or she can assume that local clocks show real time, and thus also that publications actually achieve common knowledge. This is formally shown below.

For a knowledge-based protocol to behave as if clocks show real time, it should assume that clocks show real time; that is, that only real-time histories are possible. Therefore the protocol should compute processor knowledge with respect to \(S_P(\lambda, r)\).

Note that \(S_P(\lambda, l)\) and \(S_P(\lambda, r)\) (systems with publications) have the same relationship as \(S(\lambda, l)\) and \(S(\lambda, r)\) (systems without publications). In particular, the following theorem (similar to Theorem 2) holds.

**Theorem 15**: \(\Gamma \in S_P(\lambda, l) \Rightarrow RT(\Gamma) \in S_P(\lambda, r)\).

**Proof**: Let \(\Gamma \in S_P(\lambda, l)\). Since \(RT(\Gamma)\) is a real-time history, it suffices to show that \(RT(\Gamma) \in S_P(\lambda, r)\). By Theorem 14, \(\Gamma\) satisfies the publication axioms. Since these axioms do not refer to clock history functions, \(RT(\Gamma)\) must satisfy them also. The remainder of the proof is similar to that of Theorem 2.

Consider \(\Gamma \in S_P(\lambda, l)\). Processors cannot distinguish \(\Gamma\) and \(RT(\Gamma)\). because \(\Gamma \sim RT(\Gamma)\). Since \(RT(\Gamma) \in S_P(\lambda, r)\), processors in \(\Gamma\) can use \(S_P(\lambda, r)\) as admissible set without being able to detect any inconsistency.

27
Lemma 16: Suppose that $\Gamma \in S_P(\Lambda, L)$ is consistent with knowledge-based protocol $\Pi$ and admissible set $S_P(\Lambda, R)$. Then $RT(\Gamma) \in S_P(\Lambda, R)$ and is also consistent with $\Pi$ and $S_P(\Lambda, R)$.

Proof: $RT(\Gamma) \in S_P(\Lambda, R)$ from Theorem 15 above. $\Gamma$ is consistent with $\Pi$ and $S_P(\Lambda, R)$; that is, $\forall i \in P \forall c \in V_i[\eta_i(c) = \Pi_i(c, \sigma_i(c), F_i([c, \sigma_i(c), \tau_i(c)], S_P(\Lambda, R)))]$. This does not refer to the clock history functions, the only difference between $\Gamma$ and $RT(\Gamma)$; thus $RT(\Gamma)$ is also consistent with $\Pi$ and $S_P(\Lambda, R)$.

We can now formally show that publications can be used as if they achieve common knowledge:

Theorem 17: Let $\Sigma$ be a logical specification. Suppose that knowledge-based protocol $\Pi$ satisfies $\Sigma$ when run in an asynchronous system with real-time clocks and publications (using $S_P(\Lambda, R)$ as admissible set). Then, again using $S_P(\Lambda, R)$ as admissible set, $\Pi$ also satisfies $\Sigma$ when run in an asynchronous system with our logical clocks and our implementation of publications.

Proof: We are given that any $\Gamma \in S_P(\Lambda, R)$ consistent with $\Pi$ and $S_P(\Lambda, R)$ satisfies $\Sigma$. We want to show that any $\Gamma \in S_P(\Lambda, L)$ consistent with $\Pi$ and $S_P(\Lambda, R)$ also satisfies $\Sigma$. Consider such a $\Gamma \in S_P(\Lambda, L)$. From Lemma 16, $RT(\Gamma) \in S_P(\Lambda, R)$ and is consistent with $\Pi$ and $S_P(\Lambda, R)$. From our assumption, $RT(\Gamma)$ satisfies $\Sigma$. Since $\Gamma \sim RT(\Gamma)$ and $\Sigma$ is logical, $\Gamma$ also satisfies $\Sigma$ and we are done.

Therefore, provided that the problem specification is logical, one can derive a knowledge-based protocol for an asynchronous system, $S(\Lambda, \Lambda)$, by first making the simplifying assumption that processors have real-time clocks and can use publications to achieve common knowledge. Then, one can run this protocol with our logical clocks and publications (using $S_P(\Lambda, R)$ as admissible set). Theorem 17 ensures that the protocol still satisfies specification $\Sigma$.

8.2 An Application

We consider mutual exclusion in a distributed database. Different objects in the database reside on different processors (owners). Before a processor can perform an operation on an object, it must first ensure that no other processor will do so concurrently. One way to implement this mutual exclusion is by acquiring a lock for that object. Owners grant only one lock per object, and processors release locks when they no longer need them.

We first consider a simple solution to this problem. When a processor needs a set of locks, it sends requests to the owners, and waits for all locks to be granted. If an owner receives a request for a free lock, it grants that lock to the requesting processor. If the requested lock
is not free, the requesting processor is placed on a queue of processors awaiting that lock. When a processor is done with a lock, it releases it by notifying the owner.

Unfortunately, this solution is prone to deadlock. Suppose there are four processors, 1, 2, 3, and 4. Object A resides at processor 1, and B at processor 2. Suppose that processors 3 and 4 both request locks for A and B. Processor 1 first receives the request for A from 3, and 2 first receives the request for B from 4. Thus processor 3 is granted the lock for A and 4 the lock for B. The system is now deadlocked. Processors 3 and 4 are "blocked" awaiting locks from 2 and 1, respectively, and these locks cannot be granted until they are released by 4 and 3. This type of deadlock can occur in distributed resource allocation [Brac84].

We seek a solution that prevents such deadlocks. If we assume that processors have real-time clocks and publications that achieve common knowledge, we can easily find such a solution. With this assumption, we saw that the ability to publish a fact is equivalent to writing that fact in a shared publication table (the contents of which is common knowledge). Hence processors can simply publish their requests for and releases of objects in that shared table. All processors see the same sequence of requests and releases, ordered by publication time. Using a common policy, each processor can use this sequence to determine whether or not it has acquired exclusive access to all objects it has requested. We give one such policy.

Object o is available at a given real time if all previously published and granted (see below) requests for o have a corresponding published release. A processor is granted a request for a set S of objects if, at the time this request is published, all the objects in S are available, and there is no conflicting request published at the same time. If a request is granted, the processor gains exclusive access to all requested objects.

The specification of deadlock-free mutual exclusion is logical since it does not mention real time. Therefore, Theorem 17 guarantees that the above solution is still correct when run with our logical clocks and publication mechanism.

8.3 Providing the Abstraction of Common Knowledge in Synchronous Systems

The arguments given in Section 8.1 follow when considering, in place of the asynchronous systems $S_P(A, R)$ and $S_P(A, L)$, the synchronous systems $S_P(s\Delta', R)$ and $S_P(l\Delta, s(\rho, \epsilon))$, where $\Delta' = \epsilon + (1 + \rho)\cdot\Delta$. We can prove synchronous versions of Theorem 15, Lemma 16, and Theorem 17:

**Theorem 18:** $\Gamma \in S_P(l\Delta, s(\rho, \epsilon)) \Rightarrow RT(\Gamma) \in S_P(s\Delta', R)$, where $\Delta' = \epsilon + (1 + \rho)\cdot\Delta$.

**Proof:** Similar to the proofs of Theorems 7 and 15. $\square$

29
Lemma 19: Suppose that $\Gamma \in S_p(\Delta, s(\rho, \epsilon))$ is consistent with knowledge-based protocol $\Pi$ and admissible set $S_p(s\Delta', r)$. Then $RT(\Gamma) \in S_p(s\Delta', r)$ and is also consistent with $\Pi$ and $S_p(s\Delta', r)$, where $\Delta' = \epsilon + (1 + \rho) \cdot \Delta$.

Proof: Similar to that of Lemma 16.

Theorem 20: Let $\Sigma$ be a logical specification. Suppose that knowledge-based protocol $\Pi$ satisfies $\Sigma$ when run in $S_p(s\Delta', r)$, a synchronous system with real-time clocks and publications, using $S_p(s\Delta', r)$ as admissible set. Then, again using $S_p(s\Delta', r)$ as admissible set, $\Pi$ also satisfies $\Sigma$ when run in $S_p(\Delta, s(\rho, \epsilon))$, a synchronous system with logical communication and our implementation of publications, where $\Delta' = \epsilon + (1 + \rho) \cdot \Delta$.

Proof: This follows along lines similar to that of Theorem 17.

Given a synchronous system, $S(s\Delta, s(\rho, \infty))$, we showed how to implement $S_p(\Delta, s(\rho, \epsilon))$ for some $\epsilon$. By Theorem 20, $S_p(\Delta, s(\rho, \epsilon))$ provides the abstractions of real-time clocks and the ability to achieve common knowledge, that is, the abstraction that processors are running in the system $S_p(s\Delta', r)$.

9 Discussion and Conclusions

Access to real-time clocks and the ability to achieve common knowledge simplifies the derivation and the proof of distributed algorithms. Unfortunately, in most practical systems, including the synchronous and asynchronous systems considered in this paper, perfectly synchronized real-time clocks cannot be implemented and common knowledge cannot be achieved [Halp84]. The impossibility of achieving common knowledge stems from the inability of processors to perform actions at the same real time.

For both asynchronous and synchronous systems, we ensured that local clocks reflect the potential causality between events defined by Lamport [Lamp78]. We showed that our clocks can be used as if they were perfectly synchronized real-time clocks when solving a large class of problems (those with logical specifications). We then used these clocks to implement a broadcast primitive and proved that this primitive can be used as if it achieved common knowledge (for this same class of problems). Thus, with these tools, algorithms can be derived and proven correct with the simplifying assumption that processors have access to real-time clocks and a broadcast primitive that achieves common knowledge (and hence, that processors have access to some shared memory).

When considering synchronous systems, we only showed how to apply our technique to problems with logical specifications. In these systems, however, our results also apply to a
larger class of problems, i.e., problems with real time constraints. This generalization will be fully described in a future paper.

Acknowledgments

We would like to thank Joseph Halpern, Thomas Joseph, Jennifer Lundelius, Yoram Moses, Prakash Panangaden, Rohit Parikh, T.K. Srikanth, and the students in the distributed systems group at Cornell for discussing this work with us and/or commenting on earlier drafts of this paper. We would like to thank Fred Schneider for suggesting the extension to synchronous systems.

References


