On Continuously Attaining Levels of Concurrent Knowledge without Control Messages

Ajay Kshemkalyani

Dept. of Electrical Engineering and Computer Science
1120 Science and Engineering Offices
851 South Morgan Street
University of Illinois at Chicago
Chicago, IL 60607-7053, USA

E-mail: ajayk@eecs.uic.edu

January 19, 2000

Abstract

This paper examines the relationship between the size of logical clocks and the levels of concurrent knowledge that can be attained in asynchronous distributed message-passing systems. To this end, logical clocks of arbitrary dimensions are defined. For any fact ϕ on the system state, the following is shown. $(E^C)^k(\phi)$ can be attained by the processes on a continuous basis and without using any control messages if and only if logical clocks of dimension $k+1$ and a corresponding timestamping scheme are used. This establishes a lower bound of $\Omega(n^{k+1})$ on the space complexity, where n is the number of processes. An algorithm that meets this lower bound is then presented. The paper also presents an algorithm to address the following related problem. "Given a $k+1$-dimensional timestamp, identify the maximal past computation about which $(E^C)^k(\phi)$ knowledge, for some fact ϕ, can be declared in the current state." The computational complexity of this algorithm is $O(k \cdot (n^2+n))$.

Key words: Distributed computation - Knowledge - Clocks - Time - Causality

1 Introduction

Formalizing the levels of knowledge in a distributed system simplifies the analysis of the requirements of solutions to various problems and the design of various protocols to solve the problems [7, 8, 10]. It enables the determination of whether the level of knowledge attained by a given protocol is sufficient to solve a given problem. Knowledge analysis also provides a formal method for reasoning about distributed protocols and *executions* (or *runs* or *computations*) of asynchronous distributed systems where different executions of the same program can yield different equivalent executions [1]. Halpern and Fagin [8] and Halpern and Moses [10] have extensively developed the theory of knowledge analysis in distributed systems [5].

A distributed system can be modeled by a network (N, L), where N is the set of processes that communicate by asynchronous message passing over L, the set of logical communication links. *Common knowledge*, which has been proposed as a definition of agreement in distributed systems [10], is defined as follows. A process i that knows a fact ϕ is said to have knowledge $K_i(\phi)$, and if "every process in the system knows ϕ", then the system exhibits knowledge $E^1(\phi) = \bigwedge_i K_i(\phi)$. A knowledge level of $E^2(\phi)$ indicates that every process knows $E^1(\phi)$, i.e., $E^2(\phi) = E\ (E^1(\phi))$. Inductively, a hierarchy of levels of knowledge $E^j(\phi)$ $(j > 0)$ gets defined because $E^{k+1}(\phi) \Longrightarrow E^k(\phi)$. Each level in the hierarchy represents a different level of group knowledge among the processes. Common knowledge of ϕ, denoted as $C(\phi)$, is defined as the knowledge X which is the greatest fixed point of $E(\phi \wedge X)$ and is equivalent to $\bigwedge_{j \in N} E^j(\phi)$.

Common knowledge requires simultaneous action for its achievement [8, 10] and is therefore unattainable in distributed systems that communicate by asynchronous message passing (henceforth, just asynchronous distributed systems). Rather than using common knowledge which is required to be attained simultaneously in physical time, Panangaden and Taylor proposed *concurrent common knowledge* which is required to be attained simultaneously in logical time based on causality [15], and is attainable in asynchronous distributed systems [19]. Specifically, concurrent common knowledge can be attained at a "consistent cut" or possible global state [2] in the system execution. To define concurrent common knowledge, [19] first defines $P_i(\phi)$ to represent the statement "there is some consistent global state of the current execution that includes i's local state, in which ϕ is true." $E^C(\phi) = \bigwedge_i K_i P_i(\phi)$ and is attainable by the processes at a consistent global state. Likewise, higher levels of knowledge $(E^C)^k(\phi)$ are attainable by the processes at a consistent global state. Concurrent common knowledge of ϕ, denoted by $C^C(\phi)$, is defined as the knowledge X which is the greatest fixed point of $E^C(\phi \wedge X)$ and is equivalent to $\bigwedge_{j \in N} (E^C)^j(\phi)$.

Concurrent common knowledge is a necessary and sufficient condition for performing concurrent actions in asynchronous distributed systems, analogous to simultaneous actions and common knowledge in synchronous systems [19]. The form of knowledge underlying many existing protocols involves processes reaching agreement about some property of a consistent global state, defined using logical time and causality, and can be easily understood in terms of concurrent common knowledge [19]. Global snapshot algorithms, proposed in [2], are run concurrently with the underlying computation and can be used to achieve concurrent common knowledge, as shown in [19]. Snapshot algorithms which can be used to achieve concurrent common knowledge typically require $|L|$ messages and d time steps, where d is the diameter of the network.

Concurrent common knowledge subsumes all other levels of knowledge $(E^C)^j(\phi)$. Several applications need only lower levels of knowledge such as $(E^C)^1$ and it seems plausible that there exist less expensive protocols than those, such as global snapshot algorithms, used to achieve concurrent common knowledge. Specifically, applications such as checkpointing, garbage collection, mutual exclusion, debugging, and defining concurrency measures in distributed asynchronous message-passing systems use vector clocks [6, 16] which provide specific knowledge of "local" facts ϕ (equivalent to concurrent knowledge $(E^C)^0(\phi)$) for the corresponding application domain. However, vector clocks are not sufficient in other areas such as designing distributed database protocols, fault-tolerant protocols, and protocols to discard obsolete information in distributed databases [11, 21, 23]. Similarly, vector clocks are not sufficient to solve the distributed dictionary and distributed log problems [23]. For the above listed applications, it is necessary to use matrix clocks defined in [11, 21, 23]. Matrix clocks contain vector clock information about other processes' views of the system state and can therefore be used to make decisions that require such information. In these

applications, matrix clocks provide concurrent knowledge $(E^C)^1(\phi)$ about facts ϕ in the corresponding application domain.

An important characteristic of the clock protocols used to achieve $(E^C)^0(\phi)$ and $(E^C)^1(\phi)$ is that they do not use any control messages, i.e., messages apart from those sent by the application, but rather piggyback control information on the application messages as and when they are sent. This is a desirable property of a protocol because it completely eliminates extra message overhead beyond what the application requires. The clock protocols discussed above are not full-information protocols [5], yet for each of the above applications, they suffice to provide the required degree of concurrent knowledge because the clocks are defined so as to capture the property of interest.

Besides the global snapshot protocol, another algorithm that is run concurrently with the underlying computation (not run in isolation) to attain $(E^C)^1(\phi)$ is as follows. A logical ring is superimposed on the network topology. Once a fact ϕ on the system state is known to a process, the process sends a control message around the ring. This message announces ϕ and also has a counter for the round number, set to one for the first round and incremented by the initiator for each additional round. When a process receives the control message with a counter value of 2, it knows that "all processes have seen the message at least once" and can therefore infer $(E^C)^1(\phi)$. When a process receives the control message with a counter value of 3, it can infer that "all the processes have inferred $(E^C)^1(\phi)$", and can therefore support $(E^C)^2(\phi)$. By induction, when a process receives the control message with a counter value of k, it can support $(E^C)^{k-1}(\phi)$. Another algorithm to attain $(E^C)^k(\phi)$ is to have serialized broadcasts in a system that guarantees causal message ordering. Each broadcast carries a sequence number. It is easy to observe that when a process receives a broadcast with counter value k, it attains knowledge $(E^C)^k(\phi)$. For values of $k < |N|$, the above two algorithms might be more suitable than snapshot algorithms when a lower level of knowledge than concurrent common knowledge is needed. Henceforth, we will implicitly refer to the concurrent versions of levels of knowledge $(E^C)^k(\phi)$ and to concurrent common knowledge $C^C(\phi)$ as just levels of knowledge $E^k(\phi)$ and common knowledge $C(\phi)$, respectively.

We make the following observations about the drawbacks of the above protocols to achieve $E^k(\phi)$.

1. For each instance of a fact about which knowledge is to be attained, the protocols other than the clock-based protocols use control messages to propagate knowledge about the fact. The use of control messages imposes an extra overhead on the system.

2. For each fact ϕ about which knowledge has to be attained, if we do not consider clock-based protocols, there is a overhead of $O(min(k \cdot |N|, |L|))$ messages to attain $E^k(\phi)$ and $O(|L|)$ messages to attain $C(\phi)$. Applications such as discarding obsolete information and garbage collection [11, 21, 23], fault tolerance and distributed database systems, and the distributed dictionary and distributed log protocols [23] require *continuously* updated systemwide information in order to perform efficiently. Such applications require knowledge about a fact defined on the system state after each message communication event of the underlying computation be propagated on a *continuous basis*. For such applications, the existing approaches of using control messages are impractical due to the high message overhead involved each time the fact changes. Thus, existing approaches other than the clock-based approaches do not provide a good solution for applications where it is useful to have knowledge about the past computation on an *ongoing basis* as the computation progresses.

1.1 Objectives

This paper examines the feasibility of achieving levels of $E^k(\phi)$ (beyond $k = 0, 1$) and explores the possible mechanisms to achieve such knowledge in the following framework. (The parameters of this framework are not arbitrary because they aim to overcome the above drawbacks).

1. No control messages can be used. Therefore, control information must be piggybacked on computation messages.

   Note that the numerous applications listed earlier that use vector or matrix clocks to achieve $E^0(\phi)$ and $E^1(\phi)$ knowledge use only timestamps as control information to capture the essential properties

of ϕ that are relevant to the corresponding application. To achieve levels of concurrent knowledge in a specific application domain, it is reasonable to assume that the control information would need to include logical clock values and logical timestamps of events.

2. Knowledge about the (past) computation should be propagated to distribute the latest information continuously on an ongoing basis.

   This requirement is analogous to the following, already discussed, Applications such as discarding obsolete information and garbage collection [11, 21, 23], fault tolerance and distributed database systems, and the distributed dictionary and distributed log problems [23] require continuously updated systemwide information in order to perform efficiently. The applications use matrix clocks which are sufficient to attain $E^1(\phi)$ knowledge, where ϕ is the property of interest to the application domain and is captured by timestamps. The timestamps distribute information about ϕ on an ongoing basis. A similar situation holds for the numerous applications using vector clocks.

The above framework is now formalized. The *full-information protocol (FIP)* which attains common knowledge (in a synchronous system) has been defined such that "at each step (after each local event), a process broadcasts (via control messages) to other processes its local state (which captures everything it knows)" [5]. The FIP is very expensive, and does not meet our criteria of "no control messages". We now define a "no control messages protocol", the *full-information piggybacking protocol (FIPP)*, to be one in which on each computation message of the application, the local state information is piggybacked by the sender. This protocol meets the criteria of the desired framework above. This protocol is expensive in terms of the information piggybacked. We define the k-bounded-information piggybacking protocol to overcome this drawback.

Definition 1 *The* k-bounded-information piggybacking protocol (KIPP) *is such that on each computation message of the application, k-bounded state information is piggybacked by the sender j, where k-bounded information is information of the form*

$$K_j(K_{i_1}(K_{i_2} \ldots (K_{i_k}(\phi)) \ldots)), where\ i_1, i_2, \ldots i_k \in N$$

for any fact ϕ on the system state.

Just as applications that use vector clocks (for checkpointing, garbage collection, mutual exclusion, debugging, and defining concurrency measures in distributed asynchronous message-passing systems) and those that use matrix clocks (for designing distributed database protocols, fault-tolerant protocols, protocols to discard obsolete information in distributed databases, protocols for the distributed dictionary and distributed log problems) use clocks/timestamps to represent 0-bounded information and 1-bounded information, respectively, relevant to the application, we will assume that appropriate clocks/timestamps will capture k-bounded information relevant to the application.

The objective of this paper is to study the levels of knowledge that can be attained about a fact ϕ in some global state (*cut*), and the size of clocks needed to achieve these levels of knowledge using the KIPP protocol. The paper answers the following specific questions.

Problem 1 *For any fact ϕ on the system state, what is the earliest global state in which $E^k(\phi)$ is attained using the KIPP protocol?*

Problem 2 *For any fact ϕ on the system state, what are the necessary conditions on the timestamp information required to achieve and declare $E^k(\phi)$ using the KIPP protocol?*

Problem 3 *For any fact ϕ on the system state, what are the sufficient conditions on the timestamp information required to achieve and declare $E^k(\phi)$ using the KIPP protocol?*

Problem 4 *Given a timestamp of a system state, what is the maximum execution prefix about which $E^k(\phi)$, for some fact ϕ, can be declared in the current state in a system using the KIPP protocol?*

Section 2 describes the distributed system model and the existing clock protocols to maintain logical time. Section 3 defines a new α-dimensional clock whose value is used to timestamp events and messages. It then describes the levels of knowledge that can be inferred from the α-dimensional timestamps and proves necessary and sufficient conditions on the information for the attainment of knowledge of various levels in terms of the dimensionality of the clock system used. Section 4 concludes. Proofs of theorems are given in [14].

2 System model

2.1 Model

A distributed system can be modeled by a network (N, L), where N is the set of processes that communicate by asynchronous message passing over L, the set of logical communication links. Messages are reliably delivered but can be delivered out of order. The only messages sent are those that are part of the application computation, i.e., no control messages are allowed. Let $|N|$ be denoted by n and let the processes be numbered 1 to n.

The notion of the local state of a process is primitive. An event e at process i is denoted e_i and can be (i) an internal event, (ii) a message send event at which a message is sent by process i, or (iii) a message receive event at which a message is received by process i. An event causes a local state transition and can be viewed as a function from a local state to a local state. Local states and events compose the local histories as in [10, 19]. The local history of process i, denoted h_i, is a possibly infinite sequence of alternating local states - beginning with a distinguished initial state - and events. The notation s_i^j (e_i^j) refers to the jth state (event) in process i's local history which is as follows.

$$h_i = s_i^0 e_i^1 s_i^1 e_i^2 s_i^2 e_i^3 \ldots\ldots$$

The state of a process can be obtained from its initial state and the sequence of events that have occurred up to its current state. Hence the local history may be described as the following.

$$h_i = s_i^0 e_i^1 e_i^2 e_i^3 \ldots\ldots$$

We also assume the complete-history interpretation [8, 10], i.e., the local state includes a description of all past actions in the local history. Now, an equivalent description of the history is

$$h_i = s_i^0 e_i^1 e_i^2 e_i^3 \ldots\ldots$$

Formally, an asynchronous system consists of (i) a network (N, L), (ii) a set H_i of possible local histories for each process i, (iii) a set $\mathcal{A}$ of *asynchronous runs* or *executions*, or *computations*, each of which is a vector of local histories, one per process, and (iv) messages sent as part of the asynchronous run. The system follows the KIPP protocol (Definition 1) and thus, only messages sent as part of the computation exist in the system.

A given run of the distributed system identifies a poset event structure model as in [15]. Consider a poset $(H, \prec)$ where $\prec$ is an irreflexive partial ordering. $(H, \prec)$ represents the events in the system execution which are related by the causality relation. Elements of H are partitioned into local executions at each process. Each local execution h_i is a local history and is a linearly ordered set of events in process i based on the local time of occurrence of the events. The causality relation $\prec$ is the transitive closure of the following orderings. (i) Events at a process i are ordered by the linear order of h_i. (ii) Any message send event is ordered before the message receive event at the recipient process at which the corresponding message is received. We now enhance our definition of the *local history* by assuming there are a finite number of processes i and each h_i has an initial null event $\perp_i$. Let $H^{\perp}$ denote the set of initial events. We assume that $H^{\perp}$ is an antichain and that $\forall\ \perp_i\ \forall e \in (H \setminus H^{\perp})$, $\perp_i \prec e$.

A consistent global system state is the collection of states of the processes such that if the receipt of a message is recorded, then the sending of that message is also recorded [2]. For a given run, a global state

which is the union of an arbitrarily chosen local history of each process denotes an execution prefix that may not be consistent because it may record receive events for which the corresponding send events are not recorded. A (consistent) execution prefix or a *(consistent) cut* is a downward-closed subset of H.

Definition 2 *A* cut *of execution H is a subset C of H such that $H^{\perp} \subseteq C$ and such that $e \in C \implies (\forall e')(e' \prec e \implies e' \in C)$*

For a given system run, Mattern has showed that the set of all cuts, denoted $\mathcal{C}uts$, forms a lattice ordered by the subset relation "$\subset$" and the set of downward-closed cuts is its sublattice [16]. A consistent cut denotes a feasible computation. A run of a computation traces a single maximal chain in this lattice. As all feasible global states correspond to some consistent cut, henceforth, we use the term cut to denote consistent cut unless otherwise stated.

We define $S(Cut)$ to be the set consisting of the latest event at each process in cut Cut. $S(Cut)$ denotes the "surface" of cut Cut.

Definition 3 $S(Cut) =_{def} \{e_i \in Cut \mid \forall e_i' \in Cut, e_i \succeq e_i'\}$

Observe that for a given execution $(H, \prec)$, there is a bijective mapping from the set containing each cut Cut to the set containing each surface of a cut $S(Cut)$. Given a cut Cut, Cut_i (or $S_i(Cut)$) is a subset of Cut (or $S(Cut)$) that contains events at process i.

For an event e, $\downarrow e$ is the maximal set of events that happen before or equal e.

Definition 4 $\downarrow e =_{def} \{e' \mid e' \preceq e\}$

The cut $\downarrow e$ has a unique maximal event e and is downward-closed in $(H, \prec)$. As the set of all cuts forms a lattice, therefore $\bigcap_{x \in X} \downarrow x$, which we also denote by $\cap_{\Downarrow} X$, is a downward-closed cut for any set of events X. This observation will be used subsequently for cases where $X = S(Cut)$.

Lemma 1 $e \in \cap_{\Downarrow} X \iff \forall x \in X,\ e \preceq x$

From Lemma 1, it follows that $\cap_{\Downarrow} X$, the maximum set of events that causally precede every $x \in X$, represents the maximum execution prefix such that any fact in this execution prefix is known in the local state of each process after event $x \in X$.

2.2 Logic and Formal Semantics

A Kripkean interpretation of knowledge modality requires the identification of appropriate sets of possible worlds, and a family of possible relations between those worlds. The set of possible worlds in the system model is the set of possible asynchronous runs [8, 10]. The lattice of consistent cuts is the same for all possible runs of the same program although different runs may trace a different maximal chain in this lattice [12]. Formally, (a, c) denotes a cut c in run a and $S_i((a, c))$ is the state of process i after (a, c). Two cuts c and c' are indistinguishable by process i, denoted $(a, c) \sim_i (a, c')$ iff $S_i((a, c)) = S_i((a, c'))$.

Two modal operators that are indexed by process identifiers are K_i and P_i. $K_i(\phi)$ means "ϕ is true in all possible consistent states (cuts) that include process i's local state" and $P_i(\phi)$ means "ϕ is true in some consistent state (cut) of the current run that includes process i's local state". $E^C(\phi)$ is defined as $\bigwedge_{i \in N} K_i(P_i(\phi))$. Henceforth, $(E^C)(\phi)$ is referred to simply as $E(\phi)$ or $E^1(\phi)$. A knowledge level of $E^2(\phi)$ indicates that every process knows $E^1(\phi)$, i.e., $E^2(\phi) = E\ (E^1(\phi))$. Inductively, a hierarchy of levels of knowledge $E^j(\phi)$ $(j > 0)$, gets defined because $E^{k+1}(\phi) \implies E^k(\phi)$. Each level in the hierarchy represents a different level of group knowledge among the processes. Common knowledge, denoted $C(\phi)$, is defined as the knowledge X that is the greatest fixed point of $E(\phi \wedge X)$ and is equivalent to $\bigwedge_{j \in N} E^j(\phi)$.

The formal semantics are given by the satisfaction relation $\models$ as shown below and are the same as proposed by Panangaden and Taylor [19], which in turn are the same as those of Halpern and Moses [10] except that asynchronous runs are used instead of timed runs. See [10] and [19] for details.

- ϕ can be a primitive proposition or a formula using the usual logical connectives on primitive propositions and the P, K, and C operators.
- $(a, c) \models K_i(\phi) \Leftrightarrow \forall(a', c'), ((a', c') \sim_i (a, c) \Longrightarrow (a', c') \models \phi)$
- $(a, c) \models P_i(\phi) \Leftrightarrow \exists(a, c'), ((a, c') \sim_i (a, c) \wedge (a, c') \models \phi)$
- $(a, c) \models E(\phi) \Leftrightarrow \bigwedge_{i \in N} K_i(P_i(\phi))$
- $(a, c) \models E^k(\phi), k \geq 2 \Leftrightarrow \bigwedge_{i \in N} K_i(P_i(E^{k-1}(\phi)))$

The result of Lemma 1 can be represented using this logic as follows. For any cut X, $(a, X) \models E(\cap_{\Downarrow} S(X))$ indicating that all the processes know $\cap_{\Downarrow} S(X)$ after the execution of X.

2.3 Logical Clocks

Clocks are used to maintain time at each process. Neiger and Toueg showed that for applications not dependent on real time, it is adequate to replace the absolute time clock function by a logical time clock function [18]. Logical clocks track causality which determines the extent of the past computation that could possibly be known at any event. Formally, a clock is a function that maps events in the execution history to elements in the time domain $\mathcal{T}$, i.e., $Clk : H \mapsto \mathcal{T}$. As each $e \in H$ identifies a cut $\downarrow e$, more generally, we say that clocks map cuts or global system states in the execution history to elements in the time domain $\mathcal{T}$. Such clocks are defined as follows.

$$Clk : \mathcal{C}uts \mapsto \mathcal{T}$$

Clocks provide a quantitative identifier for cuts. When we say that an event e is assigned a clock value/timestamp, more formally we mean that the cut $\downarrow e$ is assigned that clock value/timestamp. Also, a subscripted timestamp T_i denotes that this is a timestamp of an event at process i.

The following clock systems have been proposed. Lamport defined scalar clocks with the property that $e \prec f \Longrightarrow Clk(e) \prec Clk(f)$ [15]. Such clocks have the size of one integer. Mattern [16] and Fidge [6] independently formalized vector clocks which have the property that $e \prec f \Longleftrightarrow Clk(e) \prec Clk(f)$. A vector clock has size n. This is the minimum size of a clock that is required to capture the above property [3]. The following rules are used to update the clock.

1. Before process i executes an event, $Clk_i[i] = Clk_i[i] + d$ $(d > 0)$
2. A message sent by i is timestamped by the clock value of the send event. When process j receives a message with timestamp T, $\forall k \in N$, $Clk_j[k] = \max(Clk_j[k], T[k])$; execute (1); deliver the message.

A canonical version of these clocks always uses the value of 1 for d. Then an alternate definition of these clocks is as follows. Assuming that the identifier of a process i is i itself, $T(e)$ is defined as follows.

Definition 5 $T(e) =_{def} \forall i \in N, T(e)[i] = |\{e_i \mid e_i \preceq e\}|$, *i.e., $T(e)[i]$ is the number of events on process i that causally precede or equal e.*

Thus, $T(e)[i] = |(\downarrow e)_i|$ when canonical clocks are used. Vector clocks are useful for applications such as distributed debugging, file consistency, implementing causal message ordering and causal memory, establishing global breakpoints, and implementing consistent checkpoints in optimistic recovery [16].

Matrix clocks contain vector clock information about other processes' views of the system and can therefore be used to make decisions that require such information. Matrix clocks are defined as an array of size $n \times n$. The following rules are used to update the clock.

1. Before process i executes an event, $Clk_i[i, i] = Clk_i[i, i] + d$ $(d > 0)$
2. A message sent by i is timestamped by the clock value of the send event. When process j receives a message with timestamp T,
   $\forall k \in N$, $Clk_j[j, k] = max(Clk_j[j, k], T[i, k])$
   $\forall k \in N \setminus \{j\}, \forall l \in N$, $Clk_j[k, l] = max(Clk_j[k, l], T[k, l])$
   execute (1); deliver the message.

With matrix clocks, $T(e)[i]$ is the vector timestamp of $S_i(\downarrow e)$. A canonical version of matrix clocks always uses the value of 1 for d. Here, $T(e)[i,j] = |S_j(\downarrow S_i(\downarrow e))|$ when canonical clocks are used. Matrix clocks have been used to design distributed database protocols and fault-tolerant protocols, and protocols to discard obsolete information in distributed databases [11, 21, 23]. They have also been used to solve the distributed dictionary and distributed log problems [23].

Vector clocks can be used to assign timestamps to cuts. For a cut Cut, we define its timestamp $T(Cut)$ such that the ith component of the timestamp is the ith component of the timestamp of $S_i(Cut)$, viz., the maximum of the ith components of the timestamps of all the events in Cut that occur at process i. This definition assumes that the cut is consistent.

Definition 6 $T(Cut) =_{def} \forall i \in N, T(Cut)[i] = T(S_i(Cut))[i]$

The vector timestamp of a cut identifies the number of events at each process i in the cut when canonical clocks are used.

Lemma 2 *The timestamp of cut* $\bigcap_{x \in X} \downarrow x$ *, denoted* $T(\cap_{\Downarrow} X) =_{def} \forall i \in N, T(\cap_{\Downarrow} X)[i] = min_{x \in X}(T(\downarrow x)[i])$

A cut is a collection of events, and is also termed a nonatomic event [13]. The definition of the timestamp of a nonatomic event is taken from [13]. Lemma 2 will be used in Section 3 by setting the nonatomic event to contain the events in the surface of a cut. As there is a bijective mapping from the set containing each cut Cut and the set containing each surface of a cut $S(Cut)$, the timestamp of $S(Cut)$ is defined to be the timestamp of the corresponding Cut. Henceforth, any reference to $T(Cut)$ also refers to $T(S(Cut))$ and vice-versa.

3 Attaining knowledge using clocks

Matrix clocks provide more information than vector clocks which in turn provide more information than scalar clocks. This section (i) defines clocks of higher dimensions, and (ii) analyzes the amount of knowledge in terms of the E operator that can be captured by such clocks.

Definition 7 *A α-dimension clock Clk^α defines the mapping,*

$$Clk^\alpha : \mathcal{C}uts \mapsto I\!N^{n^\alpha}$$

(i.e., Clk^α is an α dimensional array of integers, where each dimension is of size n), satisfying the following properties

1. *The local clock component $Clk^\alpha[j, \ldots, j]$ must be incremented by a natural number for each local event occurrence at j.*
2. *Any element $Clk^\alpha(e_j)[i_1, i_2, \ldots, i_\alpha]$ indicates the maximum scalar clock value ϕ_{i_α} at i_α such that $K_j(K_{i_1}(K_{i_2}(K_{i_3}(\ldots K_{i_\alpha}(\phi_{i_\alpha})\ldots))))$.*
   With canonical clocks, $d = 1$ and $Clk^\alpha(e)[i_1, i_2, \ldots, i_\alpha] = |S_{i_\alpha}(\ldots \downarrow S_{i_3}(\downarrow S_{i_2}(\downarrow S_{i_1}(\downarrow e)))\ldots)|$

The value of Clk^α assigned as a timestamp is denoted T^α. We now formalize the notations used to denote multidimensional clocks and timestamps.

$T^\alpha[i]$ is a timestamp of dimension $(\alpha - 1)$. This is alternately represented as $T^\alpha[i, \cdot]$. The timestamp $T^\alpha[\underbrace{i, j, k, \ldots, l}_{\beta times, \beta \geq 0}, \cdot]$ is a timestamp of dimension $(\alpha - \beta)$, where $\cdot$ represents a β-dimensional timestamp.

Using α-dimensional clocks, each event can be assigned timestamp T^β, where $0 \leq \beta \leq \alpha$. The timestamp T^β would be $T^\alpha[\underbrace{i, i, \ldots, i}_{\alpha - \beta times}, \underbrace{\cdot}_{\beta, \beta \geq 0}]$, where $\cdot$ represents a β-dimensional timestamp.

If T^α is the timestamp of event e_i, then the scalar (i.e., 0-dimensional) timestamp of the event is $T^\alpha[i, i, \ldots, i]$. The rules to update the α-dimensional clock are given in Figure 1.

1. Before process i executes an event, $Clk_i^\alpha[i,i,\ldots,i] = Clk_i^\alpha[i,i,\ldots,i] + d$ $(d > 0)$

2. A message sent by i is timestamped by the clock value of the send event. When process j receives a message with timestamp T^α,

   (a) **for** $\beta = 1$ **to** $\alpha - 1$ **do**
   $\forall q_1 \in N \setminus \{j\}, \forall q_2, q_3, \ldots, q_\beta \in N,$
   $Clk^\alpha[j,\ldots,j,q_1,q_2,\ldots,q_\beta] = max(Clk^\alpha[j,\ldots,j,q_1,q_2,\ldots,q_\beta], T^\alpha[i,\ldots,i,q_1,q_2,\ldots,q_\beta])$

   (b) $\forall q_1 \in N \setminus \{j\}, \forall q_2, \ldots, q_\alpha \in N,$
   $Clk^\alpha[q_1,q_2,\ldots,q_\alpha] = max(Clk^\alpha[q_1,q_2,\ldots,q_\alpha], T^\alpha[q_1,q_2,\ldots,q_\alpha])$

   (c) execute(1); deliver the message.

Figure 1: Protocol to update α-dimension clocks

1. Before process i executes an event, $Clk_i^\alpha[i,i,\ldots,i] = Clk_i^\alpha[i,i,\ldots,i] + d$ $(d > 0)$

2. A message sent by i is timestamped by the clock value of the send event. When process j receives a message with timestamp T^α,

   (a) **for** $\beta = 1$ **to** $\alpha - 1$ **do**
   $\forall q_1 \in N \setminus \{j\}, \forall q_2, q_3, \ldots, q_\beta \in N,$
   if $T^\alpha[i,\ldots,i,\underbrace{q_1,q_1,\ldots,q_1}_{\beta\ times}] > Clk^\alpha[j,\ldots,j,\underbrace{q_1,q_1,\ldots,q_1}_{\beta\ times}]$
   then $Clk_j^\alpha[\underbrace{j,\ldots,j}_{\alpha-\beta\ times},q_1,\cdot] = T^\alpha[\underbrace{i,\ldots,i}_{\alpha-\beta\ times},q_1,\cdot]$

   (b) $\forall q_1 \in N \setminus \{j\},$
   if $T^\alpha[q_1,q_1,\ldots,q_1] > Clk_j^\alpha[q_1,q_1\ldots,q_1]$ **then** $Clk_j^\alpha[q_1,\cdot] = T^\alpha[q_1,\cdot]$

   (c) execute(1); deliver the message.

Figure 2: Efficient protocol to update α-dimension clocks

Theorem 1 *The protocol in Figure 1 implements the α-dimensional clock specification of Definition 7.*

The protocol in Figure 1 which implements the α-dimensional clock specification of Definition 7 can be simplified as follows. Step 2(a) can be simplified by observing that the entire $Clk_j^\alpha[j,\ldots,j,q_1,\cdot]$ needs to be replaced by $T^\alpha[i,\ldots,i,q_1,\cdot]$ if and only if the timestamp on the incoming message indicates a more recent state of q_1 than does the local clock value. Step 2(b) can be simplified by observing that the entire $Clk_j^\alpha[q_1,\cdot]$ needs to be replaced by $T^\alpha[q_1,\cdot]$ if and only if the timestamp on the incoming message indicates a more recent state of q_1 than does the local clock value. The simplified protocol is shown in Figure 2.

Observe that the size of each clock and timestamp of dimension k is n^k integers. The clock size as well as the amount of information sent as timestamps can be reduced by making certain assumptions on the communication pattern and/or the topology, using schemes similar to [17, 22]. However, the order of magnitude of the size of clocks and the piggybacked timestamp on messages in the general case without making any simplifying assumptions remains $O(n^k)$. Certain optimizations of matrix clocks that do not compute the accurate matrix clock but some approximation of it were described in [11, 23]. Similar techniques can be used for α-dimensional clocks if accuracy can be sacrificed. However, for maintaining accurate clocks, the order of magnitude of the size of clocks and the piggybacked timestamp on messages remains $O(n^k)$.

The α-dimensional timestamp of a cut is defined as follows.

(1) Initially:
(2) T^1_ϕ represents computation prefix ϕ;
(3) $TS^1 = T^1_\phi$;

(4) Timestamp TS^1 Compute_State(Timestamp T^1_ϕ, TS^1; Level k)
(5) {
(6) **for** $lvl = 1$ **to** k **do**
(7) $\forall p \in N$ **do**
(8) identify earliest event $e_p \mid T^1(e_p) \geq TS^1$;
(9) $T'^1[p] = T^1(e_p)[p]$;
(10) $\forall p \in N$ **do**
(11) $TS^1[p] = max(T^1(e_1[p]), T^1(e_2[p]), \ldots, T^1(e_n[p]))$;
(12) % **invariant** $(a, TS^1) \models E^{lvl}(T^1_\phi) \bigwedge \nexists\, TS'^1 \mid (TS'^1 < TS^1 \wedge (a, TS'^1) \models E^{lvl}(T^1_\phi))$
(13) }

Figure 3: Protocol to compute the earliest system state in which $E^k(\phi)$ is achievable

Definition 8 $T^\alpha(Cut) =_{def} \forall i \in N, T^\alpha(Cut)[i, \cdot] = T^\alpha(S_i(Cut))[i, \cdot]$

The α-dimensional timestamp of a cut identifies the $(\alpha - 1)$-dimensional timestamp of the latest event at each process in that cut.

Lemma 3 *The timestamp of cut* $\bigcap_{x \in X} \downarrow x$ *, denoted* $T^\alpha(\cap_\Downarrow X) =_{def} \forall i \in N, T^\alpha(\cap_\Downarrow X)[i, \cdot]$ *is the* $(\alpha - 1)$*-dimensional timestamp* $T^\alpha(x')[i, \cdot]$*, where* $T^\alpha(x')[i, i, \ldots, i] = min_{x \in X}(T^\alpha(x)[i, i, \ldots, i])$

Lemma 3 gives a way to implement the test for Lemma 1. It will be used in Theorem 3 to identify the minimum computation prefix (cut) at which knowledge $E^k(\phi)$ is attained about some given cut ϕ, and in Theorem 4 to identify the maximum computation prefix ϕ (cut) about which knowledge $E^k(\phi)$ has been attained at a given cut Cut.

The objective of this paper is to study the relationships among the levels of knowledge that can be attained about a fact ϕ, the maximum execution prefix on which the ϕ is specified, the execution prefix at which the desired level of knowledge is attained, and the size of clocks and timestamps needed to achieve this knowledge using the KIPP protocol. As there is a bijective mapping between the set of all states and the set of their timestamps for a clock system of a given dimension ≥ 1, therefore the truth value of a fact which is a property of a sytem state is a function of the timestamp of the global state in which it is defined. Hence, we treat facts as synonymous to the cuts on which they are defined and assign them timestamps to specify them uniquely. *As discussed and justified after Definition 1 in Section 1, the k-bounded information piggybacked on the computation messages in a KIPP protocol can be adequately represented by a timestamp to capture the property of the k-bounded information, that is of interest to any specific application running the computation.* We formalize this as the following assumption.

Assumption 1 *A fact (whose truth value is a function of the global state in which it is defined) will be uniquely identified by the timestamp of the global state.*

With the above assumption, when we need to identify the maximum possible fact ϕ such that $(a, c) \models E^k(\phi)$ at a given cut c, we focus attention on the maximum possible computation prefix Cut such that $(a, c) \models E^k(Cut)$. Likewise, when we need to identify the minimum possible computation prefix (cut) c such that $(a, c) \models E^k(\phi)$ for a given ϕ, we focus attention on the cut Cut in which ϕ is defined and identify the minimum possible computation prefix c such that $(a, c) \models E^k(Cut)$.

Theorem 2 addresses Problem 1 which seeks to identify in a system following a KIPP protocol, the earliest global state in which $E^k(\phi)$ holds, where ϕ is a given fact.

(1) Initially:
(2) T^1_ϕ represents computation prefix ϕ;
(3) $TS^1 = T^1_\phi$;

(4) Timestamp T^{k+1} Compute_Timestamp(Timestamp T^1_ϕ, TS^1; Level k)
(5) {
(6) **for** $lvl = 1$ **to** k **do**
(7) $\forall p \in N$ **do**
(8) identify earliest event $e_p \mid \forall q \in N, T^{lvl}(e_p)[q, q, \ldots, q] \geq TS^{lvl}[q, q, \ldots, q]$;
(9) $T'^{lvl}[p, \cdot] = T^{lvl}(e_p)[p, \cdot]$;
(10) $\forall p \in N$ **do**
(11) identify $r \mid T^{lvl}(e_r)[p, \ldots, p] = max(T^{lvl}(e_1[p, \ldots, p]), T^{lvl}(e_2[p, \ldots, p]), \ldots, T^{lvl}(e_n[p, \ldots, p]))$;
(12) identify $e'_p \mid T^{lvl}(e'_p)[p, \ldots, p] = T^{lvl}(e_r)[p, \ldots, p]$;
(13) $TS^{lvl+1}[p, \cdot] = T^{lvl}(e'_p)$;
(14) % **invariant** $(a, TS^{lvl+1}) \models E^{lvl}(T^1_\phi) \bigwedge \nexists\, TS'^{lvl+1} \mid (TS'^{lvl+1} < TS^{lvl+1} \wedge (a, TS'^{lvl+1}) \models E^{lvl}(T^1_\phi))$
(15) }

Figure 4: Protocol to compute the system state/timestamp needed to achieve and declare $E^k(\phi)$

Theorem 2 *Given a fact ϕ, the earliest global state in which $E^k(\phi)$ is attained in a system following the KIPP protocol is given by the protocol in Figure 3.*

The protocol in Figure 3 computes the earliest state TS^1 such that $(a, TS^1) \models E^k(\phi)$. This protocol must be executed a posteriori to the computation. Having computed TS^1, there is no way to compute ϕ itself using TS^1. This drawback is addressed by the protocol in Figure 4 that is similar to the protocol in Figure 3 and also returns the earliest global state (cut) in which $E^k(\phi)$ is attained. This state is specified using a timestamp TS^{k+1}. Each event in the surface of this cut is specified by a timestamp T^k. Once $E^k(\phi)$ is attained at this cut, we have by definition $\bigwedge_i K_i(P_i(E^{k-1}(\phi)))$. Let $\psi = E^{k-1}(\phi)$. Even with timestamps T^k for each event in the surface of cut TS^{k+1}, no process i is able to detect $E(\psi)$ and compute ψ. This is because all the p timestamps T^k are not available to any process and these are all necessary to detect $E(\psi)$. Thus, in order to detect $E(\psi)$ and compute ψ, each process needs added information about the events at which other processes achieve $K_i(P_i(\psi))$. Therefore, detecting or declaring a level of knowledge by a process (i) is more complex than attaining it, and (ii) occurs at a later cut than the cut where it is attained.

Theorem 3 addresses Problem 2 which states "Given a fact ϕ, what are the necessary conditions on the timestamp information required to achieve and declare $E^k(\phi)$ in a system following the KIPP protocol?" by using the protocol in Figure 4.

Theorem 3 *(Necessity Condition:) Knowledge $E^k(\phi)$, where ϕ is a fact, cannot be declared in a system following a KIPP protocol unless an α-dimension clock system, where $\alpha > k$, is used.*

The proof is by induction on k, the level of knowledge to be attained. Assume that $(a, Cut) \models \phi$, where Cut is any random cut. The induction hypothesis is: if timestamp T^{k+1} is not available, then knowledge $E^k(\phi)$, where ϕ is a fact on the greatest possible computation prefix, cannot be declared and ϕ cannot be computed in a system following the KIPP protocol. The protocol in Figure 4 is used in the proof of Theorem 3 to identify the earliest state in which $E^k(\phi)$ can be achieved and the minimum size of logical clocks needed to declare $E^k(\phi)$. From Theorem 3, it follows that the space complexity to achieve $E^k(\phi)$ for a fact ϕ using a KIPP protocol is $\Omega(n^{k+1})$.

Theorem 4 addresses Problem 3 which states "Given a fact ϕ, what are the sufficient conditions on the timestamp information required to achieve and declare $E^k(\phi)$ using a KIPP protocol?" by giving a protocol in Figure 5 that, given $E^k(\phi)$ in a state with timestamp T^β, where $\beta > k$, computes the state in which ϕ holds.

(1) Initially:
(2) $\beta > k \geq 1$; $\phi = H^{\perp}$; $attained = 0$; $\alpha = \beta$; T^{α} = Obs_T^{β};

(3) Timestamp ϕ Compute_Phi(Timestamp T^{α}; History ϕ; Level k, $attained$)
(4) {
(5) $\forall p \in N$ **do**
(6) $T_p^{\alpha-1} = T_p^{\alpha}[p, \cdot]$;
(7) $\forall p \in N$ **do**
(8) let r be such that $T_r^{\alpha-1} = min_{q \in N}(\{T_q^{\alpha-1}[p, p, \ldots, p]\})$;
(9) $T_p^{\alpha-2} = T_r^{\alpha-1}[p, \cdot]$;
(10) $T_*^{\alpha-1} = [T_1^{\alpha-2}, T_2^{\alpha-2}, \ldots, T_n^{\alpha-2}]$;
(11) % **invariant** $(a, T^{\alpha}) \models E^1(T_*^{\alpha-1}) \bigwedge \nexists\, T'^{\alpha-1} \mid (T'^{\alpha-1} > T_*^{\alpha-1} \wedge (a, T^{\alpha}) \models E^1(T'^{\alpha-1}))$
(12) $attained = attained + 1$;
(13) % **invariant** $(a, Obs\_T^{\beta}) \models E^{attained}(T_*^{\alpha-1}) \bigwedge \nexists\, T'^{\alpha-1} \mid (T'^{\alpha-1} > T_*^{\alpha-1} \wedge (a, Obs\_T^{\beta}) \models E^{attained}(T'^{\alpha-1}))$
(14) **if** $k = 1$ **then**
(15) $\phi = T_*^{\alpha-1}$;
(16) **return**(ϕ);
(17) **else**
(18) $Compute\_Phi(T_*^{\alpha-1}, \phi, k-1)$;
(19) % **invariant** $(a, T_*^{\alpha-1}) \models E^{k-1}(\phi) \bigwedge \nexists\, \phi' \mid (\phi' > \phi \wedge (a, T_*^{\alpha-1}) \models E^{k-1}(\phi'))$
(20) % **invariant** $(a, T^{\alpha}) \models E^{k}(\phi) \bigwedge \nexists\, \phi' \mid (\phi' > \phi \wedge (a, T^{\alpha}) \models E^{k}(\phi'))$
(21) % **invariant** $(a, Obs\_T^{\beta}) \models E^{attained+k-1}(\phi) \bigwedge \nexists\, \phi' \mid (\phi' > \phi \wedge (a, Obs\_T^{\beta}) \models E^{attained+k-1}(\phi'))$
(22) }

Figure 5: Protocol to compute state in which ϕ holds, given $E^k(\phi)$ in a state with timestamp T^{β}, where $\beta > k$

Theorem 4 *(Sufficiency condition:) Knowledge of level $E^k(\phi)$, where ϕ is a fact, can be declared in a system following the KIPP protocol if a β-dimension clock system, where $\beta > k$, is used.*

The proof is by construction. Figure 5 specifies a protocol to derive the maximum computation prefix ϕ about which the processes have knowledge $E^k(\phi)$, given the timestamp Obs_T^{β}, $\beta > k$. Obs_T^{β} is the timestamp of an individual event. This protocol assumes that the level of knowledge k in $E^k(\phi)$ that can be attained for a nontrivial ϕ is at most $\beta - 1$. This was formally shown in Theorem 3.

Initially, the processes have common knowledge only of $H^{\perp}$. T^{α} which is initially set to Obs_T^{β} is a working timestamp variable that denotes a cut. T^{α} is progressively decreased to progressively add levels of knowledge to what is known at cut Obs_T^{β}. The routine $Compute\_Phi$ is invoked recursively and behaves as follows. Given $T^{\alpha}(Cut)$, the loop in lines (5)-(6) computes the $(\alpha - 1)$-dimensional timestamp of $S_p(Cut)$ which is the latest event of the cut Cut at process p, $\forall p \in N$. $T^{(\alpha-1)}(S_p(Cut))$ is simply $T^{\alpha}[p, \cdot]$. Let X denote the events $S(Cut)$. The loop in lines (7)-(9) applies Lemma 3 to X to compute $\cap\!\!\downarrow X$ in two steps. It first identifies the timestamps $T^{(\alpha-2)}(S_p(\cap\!\!\downarrow X))$ for each process p. Then $T_*^{(\alpha-1)}(\cap\!\!\downarrow X)$ is simply the aggregation of the n timestamps $T^{(\alpha-2)}(S_p(\cap\!\!\downarrow X))$, as shown in line (10). By Lemma 1, $T_*^{(\alpha-1)}$ is the timestamp of the maximal prefix about which all the processes have knowledge at $X = S(Cut)$ and this can be asserted only at or after $S(Cut)$. Thus, $E(T_*^{(\alpha-1)})$ at T^{α} and we assert the invariant on line (11). This also adds one level of knowledge as available to the given initial state or event Obs_T^{β}, and we assert this in the invariant on line (13). If this is the desired level of knowledge, then we have the terminating case for the recursion and the value of $T_*^{(\alpha-1)}$ is returned (lines (14)-(16)), otherwise $Compute\_Phi$ is recursively invoked to determine the ϕ that is known at $T_*^{(\alpha-1)}$ for the remaining $k-1$ levels of knowledge (lines (17)-(18)).

Invariants on lines (11) and (13) are justified as above. These are the only two invariants encountered for the terminating case of recursion. Once the recursive call for values of $k \geq 2$ returns the value of ϕ, the invariants on lines (19)-(21) can be asserted by the following reasoning.

For $k = 2$, the invariant on line (19) is the same as the invariant on line (11) for the terminating case $k = 1$. The invariant on line (20) follows from the invariants on lines (19) and (11). The invariant on line (21) follows from the invariants on line (13) and (19).

For $k > 2$, the invariant on line (19) is the same as the invariant on line (20) for the invocation $k - 1$. The correctness reasoning for the invariants on lines (20) and (21) is the same as that for $k = 2$.

Hence, the invariants hold and ϕ is the maximum computation prefix such that $(a, Obs\_T^\beta) \models E^k(\phi)$, derived from the recursive use of Lemma 1 in this protocol.

This protocol of Figure 5 also explicitly answers Problem 4 which states "Given a timestamp T^α, what is the maximum execution prefix about which $E^k(\phi)$, where ϕ is a fact and $k \leq \alpha - 1$, can be declared in the current state if the system follows the KIPP protocol?"

Complexity: The computational complexity of the protocol in Figure 5 to derive ϕ from T^β such that $(a, T^\beta) \models E^k(\phi) \bigwedge \nexists \phi' \mid \phi' \geq \phi$, where $k < \beta$, is $O(k \cdot (n^2 + n))$. The analysis is as follows. For each invocation of $Compute\_Phi$, line (8) has n comparisons, and line (9) has one pointer assignment, leading to $n + 1$ operations. The loop (7)-(9) is executed n times, and $Compute\_Phi$ is invoked k times, leading to a total computational complexity of $O(k \cdot (n^2 + n))$. The space complexity is that of the logical clocks, which is $O(n^\beta)$ integers and meets the lower bound $\Omega(n^\beta)$ shown in Theorem 3.

4 Concluding remarks

This paper first defined the k-bounded information piggybacking protocol (KIPP) as a less expensive version of the full-information protocol (FIP) that eliminates control messages and achieves propagation of information in the system on an ongoing basis. This paper then defined logical clocks of arbitrary dimensions to capture causality relationships in distributed systems and justified that this is useful to implement a KIPP in the context of specific applications. It then proved that to achieve $E^k(\phi)$ concurrent knowledge, where ϕ is a fact, on an ongoing basis in an asynchronous message-passing distributed system using a KIPP protocol, it is necessary (Theorem 3) and sufficient (Theorem 4) to use a logical clock of $k + 1$ dimensions and a corresponding timestamping scheme for computation messages. For a system with n processes, this requires a space complexity of n^{k+1} integers for all clocks and overhead on messages. Thus, the space complexity for the problem is $\Omega(n^{k+1})$ and $O(n^{k+1})$. Given knowledge $E^k(\phi)$ in a system state identified by a $k + 1$ dimensional timestamp, the algorithm proposed to detect ϕ is given in Figure 5. It meets the necessary and sufficient conditions on the information required and has a computational complexity of $O(k \cdot (n^2 + n))$. Note that the computational complexity is less than the space complexity because information is selectively used on a dynamic basis. The algorithm has the advantages that it (i) provides the knowledge of facts on a dynamic basis and (ii) does not require control messages.

Although the space complexity of the proposed algorithm is high, it is a good alternative to known techniques under the problem specification. It is known that concurrent common knowledge can be achieved using $|L|$ control messages per fact ϕ [19]. For the given problem specification, to achieve concurrent knowledge for each and every fact (cut) on an ongoing basis will require first the enumeration of all the consistent cuts in the computation. Charron-Bost showed that there is a bijective function that maps the set of all consistent cuts to the set of all antichains of $(H, \prec)$ [4]. Hence, all the antichains of events in the computation will need to be evaluated, and then $min(k \cdot n, |L|)$ control messages will be needed to achieve E^k knowledge about each such antichain. This introduces a very high complexity of the number of control messages and computational complexity of having to enumerate all the antichains. Hence, although the space complexity of the clock system and piggybacked information on messages is exponential in the level of knowledge to be attained, it is the lower bound for the given problem specification and does not require any control messages.

References

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

[2] K. M. Chandy, L. Lamport, Finding global states of a distributed system, *ACM Transactions on Computer Systems*, 3(1): 63-75, 1985.

[3] B. Charron-Bost, Concerning the size of clocks in distributed systems, *Information Processing Letters*, 39: 11-16, 1991.

[4] B. Charron-Bost, Combinatorics and geometry of consistent cuts: Application to concurrency theory, *International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 354, 45-56, Springer-Verlag, New York/Berlin, 1988.

[5] R. Fagin, J. Halpern, Y. Moses, M. Vardi, Reasoning about Knowledge, MIT Press, 1995.

[6] C. Fidge, Timestamps in message-passing systems that preserve partial ordering, *Australian Computer Science Communications*, 10(1): 56-66, February 1988.

[7] J. Hintikka, *Knowledge and belief*, Cornell University Press, 1962.

[8] J. Halpern, R. Fagin, Modeling knowledge and action in distributed systems, *Distributed Computing*, 3(4): 139-179, 1989.

[9] J. Halpern, Y. Moses, Giude to the modal logic of knowledge and belief, *9th International Joint Conference on Artificial Intelligence*, 480-490, 1985.

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

[11] A. Krishnakumar, A. Bernstein, Bounded ignorance in replicated systems, *Proc. ACM Symposium on Principles of Database Systems*, 1991.

[12] A. Kshemkalyani, Framework for viewing atomic events in distributed computations, *Theoretical Computer Science*, 196(1-2): 45-70, April 1998.

[13] A. Kshemkalyani, Causality and atomicity in distributed computations, *Distributed Computing*, 11(4): 169-189, Oct. 1998.

[14] A. Kshemkalyani, On continuously attaining levels of concurrent knowledge without control messages, Technical Report UIC-EECS-98-8, University of Illinois at Chicago, 1998.

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

[16] F. Mattern, Virtual time and global states of distributed systems, *Parallel and Distributed Algorithms*, North-Holland, pp 215-226, 1989.

[17] S. Meldal, S. Sankar, J. Vera, Exploiting locality in maintaining potential causality, *Proc. 10th ACM Symposium on Principles of Distributed Computing*, 231-239, 1991.

[18] G. Neiger, S. Toueg, Substituting for real time and common knowledge in asynchronous distributed systems, *Distributed Computing*, 6(2): 73-94, Sept. 1992.

[19] P. Panangaden, K. Taylor, Concurrent common knowledge: Defining agreement for asynchronous systems, *Distributed Computing*, 6(2): 73-94, Sept. 1992.

[20] R. Parikh, P. Krasucki, Levels of knowledge in distributed computing, *Sadhana Journal*, 17(1): 167-191, 1992.

[21] S. Sarin, N. Lynch, Discarding obsolete information in a distributed database system, *IEEE Transactions on Software Engineering*, 13(1): 39-46, 1987.

[22] M. Singhal, A. Kshemkalyani, Efficient implementation of vector clocks, *Information Processing Letters*, 43, 47-52, August 1992.

[23] G. Wuu, A. Bernstein, Efficient solutions to the replicated log and dictionary problems, *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 232-242, 1984.