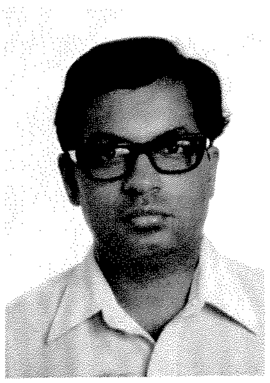
Distributed Computing (1986) 1: 40-52

DISTRIBUTED COMPUTING

How processes learn

K.M. Chandy and Jayadev Misra

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA



Jayadev Misra is a professor in the Department of Computer Sciences at the University of Texas at Austin. His primary research interests are in the area of distributed computing: specification and design of networks of asynchronous components. He believes that sound practical techniques must be based on elegant theories.

Mani Chandy is a professor of Computer Science and Electrical Engineering at the University of Texas at Austin. He is chairman of the Computer Sciences Department. His research interests are in distributed systems and performance analysis.

1 Introduction

Processes in distributed systems communicate with one another exclusively by sending and receiving messages. A process has access to its state but not to the states of other processes. Many distributed algorithms require that a process determine facts about the overall system computation. In anthropomorphic terms, processes "learn" about states of other processes in the evolution of system computation. This paper is concerned with how processes learn. We give a precise characterization of the minimum information flow necessary for a process to determine specific facts about the system.

The central concept in our study is that of *isomorphism* between system computations with respect to a process: two system computations are isomorphic with respect to a process if the process behavior is identical in both. In anthropomorphic terms, "system computations are isomorphic with respect to a process" means the process cannot distinguish between them.

Many correctness arguments about distributed systems have the following operational flavor: "I will send a message to you and then you will think that I am busy and so you will broadcast...". Such operational arguments are difficult to understand and error prone. The basis for such operational arguments is usually a "process chain": a sequence of message transfers along a chain of processes. We advocate nonoperational reasoning. The basis for nonoperational arguments is isomorphism; we relate isomorphism to process chains. Algebraic properties of system computations under isomorphism provide a precise framework for correctness arguments.

It has been proposed [3, 6] that a notion of "knowledge" is useful in studying distributed computations. In earlier works, knowledge is introduced via a set of axioms [4]. Our definition of knowledge is based on isomorphism. Our model allows us to study how knowledge

Offprint requests to: K.M.Chandy

This work was supported in part by a grant from the Office of Naval Research under N00014-85-K-0057

is "gained" or "lost". One of our key theorems states that knowledge gain and knowledge loss both require sequential transfer of information: if process q does not know fact b and later, p knows that q knows b, then q must have communicated with p, perhaps indirectly through other processes, between these two points in the computation; conversely, if p knows that q knows b and later, q does not know b then p must have communicated with q between these two points in the computation. In the first case, the effect of communication is to inform p of q's knowledge of b. Analogously, in the second case, the effect of communication is to inform q of p's intention of relinquishing its knowledge (that q knows b). Generalizations of these results for arbitrary sequences of processes are stated and proved as corollaries of a general theorem on isomorphism.

We use the results alluded to in the last paragraph for proving lower bounds on the number of messages required to solve certain problems. We show, for instance, that there is no algorithm to detect termination of an underlying computation using only a bounded number of overhead messages.

2 Model of a distributed system

A distributed system consists of a finite set of processes. A process is characterized by a set of process computations each of which is a finite sequence of events on that process. Process computations are prefix closed, i.e. all prefixes of a process computation are also process computations (of that process). An event on a process is either a *send*, a *receive* or an *internal event*. A *send* event on a process corresponds to sending a message to another process. A *receive* event on a process corresponds to reception of a message by the process. There is no external communication associated with an *internal* event. For a set of processes P, a send event by P is a send event by some component process of P to a process outside P; similarly a receive event by P denotes receipt by some process in P of a message sent from outside P. Communication among processes in P are internal events of P. We use "e is on P", for event e and process set P, to denote that e is an event on some process in P. We rule out processes which have no event in any computation. We assume that all events and all messages are distinguished; for instance, multiple occurrences of the same message are distinguished by affixing sequence numbers to them.

Let z be any sequence of events on component processes of a distributed system. The *projection* of z on a component process p, denoted by z_p, is the subsequence of z consisting of all events on p. A finite sequence of events z is a *system computation* of a distributed system means (1) for all processes p, z_p is a process computation of p and, (2) for every receive event in z, say receipt of message m by process p, there is a send event, of sending m to p, which occurs earlier than the receive in z: this send event will be called the send event *corresponding* to the receive. We leave it to the reader to show that system computations are prefix closed.

In this paper we consider a single (generic) distributed system. For instance, when we say "z is a computation" we mean that z is a computation of the distributed system considered here. We use *computation* to mean *system computation* when no confusion can arise.

Notation. We use x, y, z to denote computations, p, q for processes and P, Q for process sets; these symbols may be used with subscripts or superscripts. The concatenation of two sequences y and z will be denoted by $(y; z)$. For sequences y and z, $y \leqq z$ denotes that y is a prefix of z; in this case (y, z) denotes the suffix of z obtained by removing y from z. The empty sequence will be denoted by *null*. The symbol $=$ is used to denote equalities among sets and among predicates. The symbol $\equiv$ is used for definitions. The set of all processes in the system will be denoted by D and for any process set P, $\bar{P} = D - P$.

3 Isomorphism

We define relation $[p]$ on the set of system computations as follows.

Definition. For system computations x, y: $x[p]\,y \equiv (x_p = y_p)$.

In other words, $x[p]\,y$ means p's computation is the same in system computations x and y. In this case, we say x, y are *isomorphic with respect to* p. For a process set P, define relation $[P]$, on the system computations, as follows.

Definition. $x[P]y \equiv$ for all p in P, $x[p]y$.

Thus $x[P]y$ means that, given only the computations of processes in P we cannot distinguish x from y. From definition, $x[\{\ \}]y$, for all computations x, y where $\{\ \}$ denotes the empty set. Observe that $[P]$ is an equivalence relation.

It is convenient to represent all such isomorphism relations by an *isomorphism diagram:* an undirected labelled graph whose vertices are computations and there is an edge labelled $[P]$ between vertices x, y if P is the largest set of processes for which $x[P]y$. Observe that every vertex has a self loop labelled $[D]$ where D is the set of all processes in the system. Note that $x[D]y$, $x \neq y$, implies y is a permutation of x.

Example 1. Consider a system with two processes, p *and* q, for which part of the isomorphism diagram, showing the relationships among four system computation, is given below.

From the diagram $x[p]y$, but not $x[q]y$. This means p has the same computations in both x and y, whereas q's computations in x and y differ. Computations x and z have the same computations for both p and q; hence one is a permutation of the other. There is no direct relationship between y and w; neither $y[p]w$ nor $y[q]w$ holds. However, there is an indirect relationship between y and w because $y[p]z$ and $z[q]w$. We explore such indirect relationships next. □

Definition. Let $n>0$ and P_i be process sets, $0 \leq i \leq n$.

$x[P_0 \dots P_n]z \equiv x[P_0 \dots P_{n-1}]y$ *and* $y[P_n]z$, for some computation y.

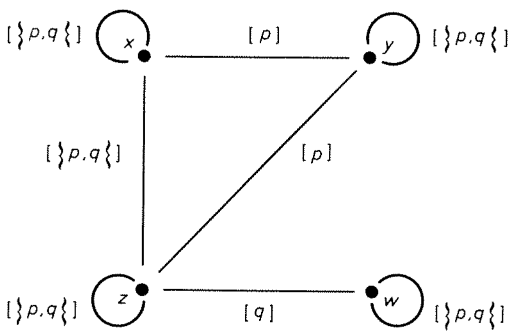


Fig. 1. An isomorphism diagram

Hence, $[PQ]=[P] \circ [Q]$ where "$\circ$" is the relational composition operator. This operator is associative (from properties of relations). In terms of the isomorphism diagram, $x[P_0 \dots P_n]z$ means there is a path from x to z whose edges are labelled with $Q_0, \dots, Q_n$, respectively, where $Q_i \supseteq P_i$, for all i.

Example 1 (contd.). We have $y[pq]w$ and $w[qp]y$. Also, trivially, $y[qp]z$, $y[qpq]z$, etc. □

We note some properties of isomorphism relations. In the following, P, $P_1, \dots, P_n$, Q, denote arbitrary process sets and x, y, z denote arbitrary computations.

1. $[P]$ is an equivalence relation.
2. (Substitution) $([\beta]=[\delta])$ *implies* $([\alpha\beta\gamma] = [\alpha\delta\gamma])$ for arbitrary sequences of process sets α, β, γ, δ.
3. (Idempotence) $[PP]=[P]$
4. (Reflexivity) $x[P_1 \dots P_n]x$
5. (Inversion) $x[P_1 \dots P_n]y = y[P_n \dots P_1]x$
6. (Concatenation) For $0<m<n$, $\exists y$: $x[P_1 \dots P_m]y$, $y[P_{m+1} \dots P_n]z$ $= x[P_1 \dots P_m P_{m+1} \dots P_n]z$
7. $[P \cup Q] = ([P] \cap [Q])$
8. $(Q \supseteq P) = ([Q] \subseteq [P])$
9. $(P=Q) = ([P]=[Q])$
10. $Q \supseteq P$ *implies* $([QP]=[P]=[PQ])$

These properties follow from properties of relations and our model. We only sketch a proof of one part of property 8:

$([Q] \subseteq [P])$ *implies* $(Q \supseteq P)$.

If $Q \not\supseteq P$ then there is a process p in $P-Q$. From our model, p has an event e in some computation $(x;e)$. Then $x[Q](x;e)$ and $\sim x[P](x;e)$. Hence $[Q] \not\subseteq [P]$.

3.1 Process chains

As noted in the introduction, the basis for many operational arguments are process chains: process p informing q which in turn, informs r etc. One of our goals is to replace such concepts by algebraic properties of system computations. In this section we show how process chains are related to isomorphism. We first define proces chains; this definition is along the lines suggested by Lamport [5].

Definition. For events e, e' in a computation z, $e \xrightarrow{z} e'$ means:

1. e' is a receive and e is the corresponding send, or
2. events e, e' are in the same process computation and ($e=e'$ or e occurs earlier than e'), or
3. there exists an event e'' such that $e \xrightarrow{z} e''$ and $e'' \xrightarrow{z} e'$.

For brevity we write $e \rightarrow e'$ when the computation z is understood from context. We will write $e_0 \rightarrow e_1 \rightarrow \dots e_{n-1} \rightarrow e_n$, as shorthand for $e_0 \rightarrow e_1$ and ... and $e_{n-1} \rightarrow e_n$. Observe that $e \rightarrow e$ for every event e in z. A computation z has a *process chain* $\langle P_0 P_1 \dots P_n \rangle$ means there exist events $e_0, e_1, \dots e_n$, not necessarily distinct, in z such that event e_i is on P_i, for all $0 \leqq i \leqq n$, and $e_0 \rightarrow e_1 \rightarrow \dots \rightarrow e_n$.

Observation 1. Any occurrence of "P" in a process chain may be replaced by "PP", or vice versa, since for any event e on P, $e \rightarrow e$.

Observation 2. Let x be a sequence consisting of a subset of events from a computation y. Suppose that for every event e in x: every e', where $e' \xrightarrow{y} e$, is also in x, and $e' \xrightarrow{x} e$. Then x is a computation.

3.2 Relationship between isomorphism and process chain

Theorem 1. (Fundamental theorem of process chains). *Let z be a computation and x a prefix of z. Let $P_1, P_2 \dots P_n$, $n \geqq 1$, be sets of processes. Then $x[P_1 P_2 \dots P_n]z$ or there is a process chain $\langle P_1 P_2 \dots P_n \rangle$ in (x,z).* □

Proof. Assuming that there is no process chain $\langle P_1 \dots P_n \rangle$ in (x,z), we show that $x[P_1 \dots P_n]z$. Proof is by induction on n. For $n=1$, absence of a process chain $\langle P_1 \rangle$ in (x,z) means that there is no event on P_1 in (x,z) and hence $x[P_1]z$. For $n>1$, we show that there is some y, $x \leqq y$, such that there is no process chain $\langle P_1 \dots P_{n-1} \rangle$ in (x,y) and $y[P_n]z$; the result then follows by inductive argument.

Let E be the subsequence of events in (x,z) consisting of the set of events $\{e | e \rightarrow e'$ where e is in (x,z) and e' is some event on $P_n\}$. Let $y = (x;\, E)$. First, we show that if $e_1 \xrightarrow{z} e_2$ and e_2 is in y then e_1 is also in y and $e_1 \xrightarrow{y} e_2$; this guarantees (from observation 2) that y is a computation. This result follows trivially when e_2 is in x. If e_2 is in (x,z) then $e_2 \xrightarrow{z} e'$, for some event e' on P_n and hence $e_1 \xrightarrow{z} e'$ and therefore e_1 is in y; the relative order between e_1, e_2 is maintained by our construction.

Next, we show that $y[P_n]z$; that is, every event on P_n that is in z is also in y. This follows trivially for events on P_n that are in x. Let e' be an event on P_n that is in (x,z). Since $e' \rightarrow e'$, e' is also in E and hence in y.

Finally, we show that there is no process chain $\langle P_1 \dots P_{n-1} \rangle$ in (x,y). If there is such a process chain, consider its last event e. According to our construction, event chain $e \rightarrow e'$ exists in (x,z), where e' is some event on P_n. Hence there is a process chain $\langle P_1 \dots P_n \rangle$ in (x,z), contradicting our assumption. □

We note that the two conditions in the last sentence of the theorem are not exclusive. Consider two computations z, z' where

z is $\langle P$ sends m to R; R receives m from P; R sends m' to Q; Q receives m' from $R \rangle$,

z' is $\langle R$ sends m' to Q; Q receives m' from $R \rangle$

In z, though there is a process chain $\langle PRQ \rangle$, there is not a "true" dependence from P to R to Q: R sends m' to Q *independent* of receiving m from P (as shown in z'). Note that *null* $[P]$ z' and z' $[Q]$ z, and hence *null* $[PQ]$ z, though (*null*, z) has a process chain $\langle PQ \rangle$.

3.3 An application of isomorphism: how to construct a computation by fusing separate ones

In this section, we show an application of isomorphism: we give a construction to "fuse" two computations to obtain a new computation, provided certain types of paths exist in the isomorphism diagram. We motivate the discussion by the following observations. Suppose $(x;E)$ and $(x;\bar{E})$ are computations where all events in E are on a process set P and all events in $\bar{E}$ are on $\bar{P}$. Then, from definition, $(x;\bar{E};E)$ and $(x;E;\bar{E})$ are also computations, because events in $E, \bar{E}$ are independent and hence may be fused in arbitrary order. A similar result appears in Fischer et al. [2]. The following lemma is a generalization of this obervation.

Lemma 1. *Let x, y, z be computations where $x \leqq y$ and $x \leqq z$. Let P, Q be such that $P \cup Q = D$, $x[P]y$ and $x[Q]z$. Then there exists a computation w where $x \leqq w$, $y[Q]w$ and $z[P]w$.* □

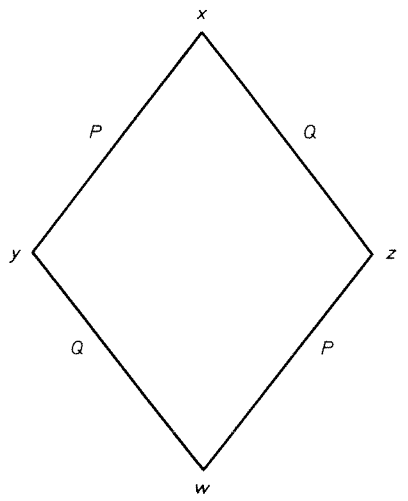


Fig. 2. Isomorphism diagram depicting fusion

The relationships among x, y, z and w are represented by the following commutative isomorphism diagram.

Proof. Let $w = x; (x, y); (x, z)$.

From the condition of the lemma, (x, y) has events only on $\bar{P}$ and (x, z) has events only on $\bar{Q}$. Since $P \cup Q = D$, $\bar{P} \cap \bar{Q} = \{\ \}$ and hence no process has events in both (x, y) and (x, z). It follows, from definition of computations, that w is a computation. Also $y[Q]w$, $z[P]w$ and $x \leqq w$, as required for proof of the lemma. □

Note that, in the construction of Lemma 1, all events from E and $\bar{E}$ were present in the fused computation. We prove a far more general result below. We show that for any two arbitrary computations y and z, the projected computations, y_P and $z_{\bar{P}}$, may be fused to form a new computation provided there is a computation x which is a prefix of both y and z, and no message sent by $\bar{P}$ in (x, y) is received by P in (x, y) and no message sent by P in (x, z) is received by $\bar{P}$ in (x, z). This makes intuitive sense: processes in P can execute all events in y given only that processes in $\bar{P}$ execute all events up to x and similarly for executions of events on $\bar{P}$ up to z. However, the statement and proof of this result are difficult without the notion of isomorphism. We note that the result may be easily generalized to fusions of arbitrary numbers of computations under similar constraints.

Theorem 2. (Fusion of computations). *Consider system computations x, y, z where $x \leqq y$ and $x \leqq z$. Let P be a set of processes such that there is no process chain,* (1) *$\langle P\bar{P} \rangle$ in (x, y) and* (2) *$\langle \bar{P}P \rangle$ in (x, z). Then there is a computation w where, $x \leqq w$, $y[\bar{P}]w$ and $z[P]w$. That is, w consists of all events on $\bar{P}$ from y and all events on P from z.* □

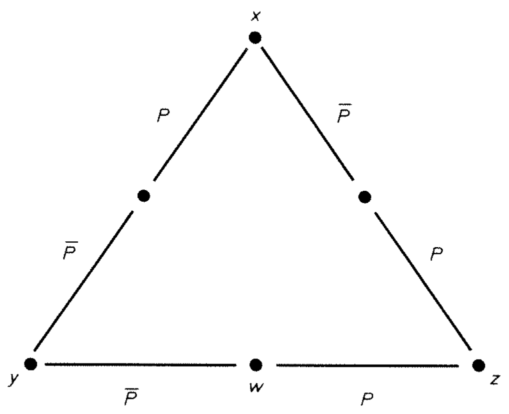


Fig. 3. Diagramatic representation of fusion theorem

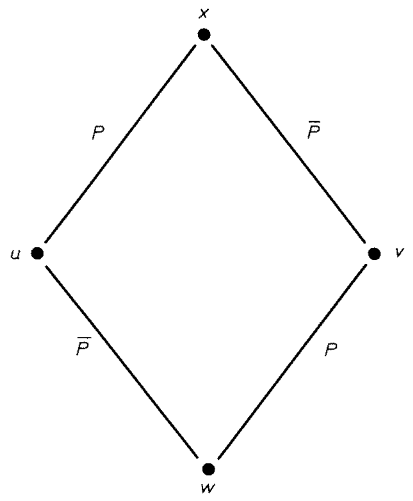


Fig. 4. Intermediate step in fusion theorem

Proof. According to Theorem 1, absence of process chains as given in this theorem means that, $x[P\bar{P}]y$ and $x[\bar{P}P]z$.

The theorem asserts the existence of the isomorphism diagram in Fig. 3. To prove that such a w exists, label the intermediate point between x, y as u and between x, z as v in this figure. Now we apply Lemma 1 to x, u, v to obtain a w, as given in Fig. 4.

Now $u[\bar{P}]y$ and $u[\bar{P}]w$; hence $y[\bar{P}]w$. Similarly $z[P]w$. This proves the theorem. Relationships among x, y, z, u, v, w are shown in Fig. 5. □

The fusion theorem is used later to obtain lower bounds on the number of messages required to solve certain problems.

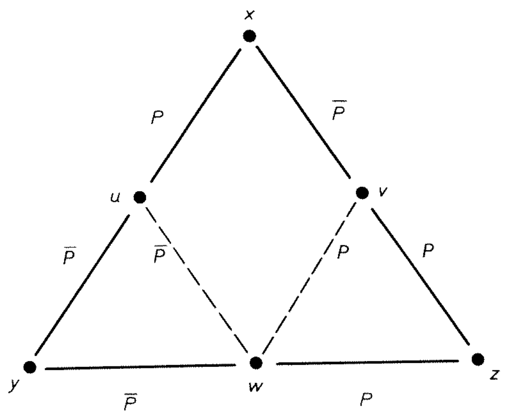


Fig. 5. Isomorphism diagram depicting proof of fusion theorem

3.4 Semantics of event types in terms of isomorphism

We now use isomorphism to state and derive some important facts about various types of events. First, note that a process carries out an internal event or sends a message depending on its own computation alone. Therefore, if a process takes such a step in a computation x, it will also do so in y, if x, y are isomorphic with respect to this process. An analogous result holds for internal and receive events. The following principle, which states these facts formally, may be proven from the definition of system computation.

Principle of computation extension:

Let e be an event on P.
1. e is an internal or send event: ($x[P]y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation.
2. e is an internal or receive event: $(x;e)[P]y$ *implies* $(y-e)$ is a computation, where $(y-e)$ is the sequence obtained by deleting e from y. □

Note. In (1), $(x;e)[P]$ $(y;e)$ and in (2), $x[P](y-e)$.

Corollary. *Let e be a receive event on P and let the corresponding send event be on Q.*

($x[P\cup Q]y$ and $(x;e)$ is a computation) implies $(y;e)$ is a computation. □

Proof. e is an internal event of $P\cup Q$. □

The following theorem follows from the principle of computation extension.

Theorem 3. *Let $(x;e)$ be a computation where e is an event on P.*

Case 1. e is a receive:

for every z: $(x;e)$ $[P\bar{P}]z$ *implies* $x[P\bar{P}]z$

Case 2. e is a send:

for every z: $x[P\bar{P}]$ z *implies* $(x;e)[P\bar{P}]z$

Case 3. e is an internal event:

for every z: $(x;e)[P\bar{P}]z=x[P\bar{P}]z$ □

Proof. We will prove only Case 2; other cases are similarly proven.

$x[P\bar{P}]z$ *implies* there exists $y, x[P]y$ *and* $y[\bar{P}]z$.

From principle of computation extension, $(y;e)$ is a computation and $(x;e)$ $[P]$ $(y;e)$. Also, $(y;e)[\bar{P}]y$. Hence, $(x;e)[P\bar{P}]z$, and therefore, $(x;e)$ $[P\bar{P}]z$. □

This theorem captures the intuitive notion that the set of possible computations, isomorphic with respect to P, can only shrink in size as a result of a reception as computations which do not include the corresponding send are ruled out. Similarly, the set of possible computations, isomorphic with respect to P cannot shrink as a result of a send: after the send, additional computations which accept the message sent are isomorphic while all prior isomorphic computations remain isomorphic. An internal event can neither expand nor shrink the set of isomorphic computations.

4 Knowledge

As we have remarked earlier, predicates of the type *P knows b at x* may be defined using isomorphism. We explore properties of such predicates in our model. We show that they satisfy the "knowledge axioms" as given in [3, 6]. We prove a general result which shows that certain forms of knowledge can only be gained or lost in a sequential fashion along a chain of processes. That is, if b is *false* for a computation and later, P_1 *knows* P_2 *knows* ... P_n *knows* b (this represents knowledge gain), then there is a process chain $\langle P_n P_{n-1} \dots P_1\rangle$ between these two points of the computation. Conversely, if P_1 *knows* P_2 *knows* ... P_n *knows* b and later, b is *false* (this represents knowledge loss), then there is a process chain $\langle P_1 P_2 \dots P_n\rangle$ between these two points of the computation.

Crucial to our work is the notion of *local predicates*: a predicate local to p can change in value only as a result of events on p. We show that local predicates play a key role in understanding knowledge predicates.

4.1 Knowledge predicates

Let b denote a predicate on system computations and "b *at* x" its value for computation x. Our predicates are *total*, i.e. for each x, b *at* x is either *true* or *false*. We furthermore assume that $x[D]y$ *implies* (b *at* $x=b$ *at* y) for every predicate b. Thus predicate values depend only upon computations of component processes and not on the way independent events are ordered in a linear representation of the computation. A predicate c is a *constant* means c *at* $x=c$ *at* y, for all computations x, y. We now define (P *knows* b) *at* x.

Definition. (P *knows* b) *at* $x=$
for all y: $x[P]y$: b *at* y

Note that b may itself be a predicate of the form Q *knows* b' in the above definition. We next note some facts about knowledge predicates. In the following, x, y are arbitrary computations, b, b' are arbitrary predicates and P, Q are arbitrary sets of processes. All facts are universally quantified over all computations. We use the convention that P *knows* Q *knows* b *at* x is to be interpreted as (P *knows* (Q *knows* b)) *at* x.

1. P *knows* b *at* x = for all y: $x[P]y$: P *knows* b *at* y
2. $x[P]y$ *implies* [P *knows* b *at* x = P *knows* b *at* y]
3. (P *knows* b) *implies* ($P \cup Q$ *knows* b)
4. (P *knows* b) *implies* (b)
5. (P *knows* b) *or* ($\sim P$ *knows* b)
6. (P *knows* b) *and* (P *knows* b') = P *knows* (b *and* b')
7. ((P *knows* b) *or* (P *knows* b')) *implies* (P *knows* (b *or* b'))
8. (P *knows* $\sim b$) *implies* ($\sim P$ *knows* b)
9. ((P *knows* b) *and* (b *implies* b')) *implies* (P *knows* b')
10. P *knows* P *knows* b = P *knows* b
11. P *knows* $\sim P$ *knows* $b = \sim P$ *knows* b
12. P *knows* c *or* P *knows* $\sim c$, for any constant c.

These facts are easily derivable from the definition of *knows*. We give a proof of (11), whose validity in other domains have been questioned on philosophical grounds [3].

Lemma 2. P *knows* $\sim P$ *knows* $b = \sim P$ *knows* b. □

Proof. P *knows* $\sim P$ *knows* b *at* x
= for all y: $x[P]y$: $\sim P$ *knows* b *at* y,
from definition
= for all y: $x[P]y$: there exists z: $y[P]z$: $\sim b$ *at* z,
from definition
= there exists z: $x[P]z$: $\sim b$ *at* z,
since $[P]$ is an equivalence relation
= $\sim P$ *knows* b *at* x. □

There are situations where multiple levels of knowledge such as, P *knows* Q *knows* b, are useful. For instance, consider a *token bus* which is a linear sequence of processes among which a token is passed back and forth; processes at the left or right boundary have only a right or left neighbor to whom they may pass the token; other processes may send it to either neighbor. There is only one token in the system and initially it is at the leftmost process. Consider a token bus with five processes labelled p, q, r, s, t from left to right. When r holds the token,

r *knows* ((q *knows* (p does not hold the token)) *and*
(s *knows* (t does not hold the token)))

Relations of the form $[PQ]$, with multiple process sets, arise from predicates with multiple occurrence of *knows*;
For instance:

p *knows* q *knows* b *at* z
= for all y: $x[p]y$: q *knows* b *at* y
= for all y: $x[p]y$: (for all z: $y[q]z$: b *at* z)
= for all z: $x[p\,q]z$: b *at* z

4.2 Local predicates

Let b be a predicate on system computations, and P a set of processes. We define a predicate P *sure* b as follows.

Definition. (P *sure* b) *at* $x \equiv$((P *knows* b) *at* x *or* (P *knows* $\sim b$) *at* x).

In other words (P *sure* b) *at* x means that P knows the value of b *at* x.

We define *unsure* as negation of *sure*.

Definition. P *unsure* $b \equiv \sim P$ *sure* b.

Hence, (P *unsure* b) *at* x=[($\sim P$ *knows* b) *at* x *and* ($\sim P$ *knows* $\sim b$) *at* x].

Definition. b is *local* to $P \equiv$for all x: (P *sure* b) *at* x.

That is, the value of b is *always* known to P. Local predicates capture our intuitive notion of a predicate whose value is controlled by the actions of processes to which it is local.

We note the following facts about local predicates; in the following, b is an arbitrary predicate and P, Q are arbitrary sets of processes.

1. (b *is local to* P *and* $x[P]y$) *implies* (b *at* $x=b$ *at* y)
2. b *is local to* P *implies* ($b=P$ *knows* b)
3. b *is local to* $P=(\sim b)$ *is local to* P.
4. b *is local to* P *implies* [Q *knows* $b=Q$ *knows* P *knows* b]
5. (P *knows* b) *is local to* P.
6. b *is local to* P *and* b *is local to* Q *and* P, Q are disjoint *implies* b is a constant.
7. b is a constant *implies* b *is local to* P.
8. (P *sure* b) *is local to* P.

Proof of (1) follows from definition of knowledge and local predicates. (2) and (3) follow trivially. (4) follows from Q *knows* b *at* $x=$ for all y: $x[Q]y$: b *at* $y=$ for all y: $x[Q]y$: P *knows* b *at* y (since b is local to P) $=Q$ *knows* P *knows* b *at* x. (5) follows from, (P *knows* P *knows* b or P *knows* $\sim P$ *knows* b) $=$ (P *knows* b or $\sim P$ *knows* b) $=$ *true*. Proof of (6) is important and hence is given below as a lemma. (7) and (8) are trivially proven from definition.

Lemma 3. *b is local to disjoint sets P, Q implies b is a constant.* □

Proof. We show that b *at* $x=b$ *at null*, for all x. Proof is by induction on length of x.

b *at null* $=b$ *at null*.
b *at* $(x;e)=b$ *at* x, because event e is not on P or e is not on Q, and hence $(x;e)[P]x$ or $(x;e)[Q]x$; then the result follows from property (1). □

For a system of processes, b *is common knowledge* is defined as the greatest fix point of the following equation.

b *is common knowledge* $\equiv$ b *and* (p *knows* b) *is common knowledge*, for all processes p. Intuitively, b *is common knowledge* means b is *true*, every process *knows* b, every process *knows* that every process *knows* b, etc.

Halpern and Moses [3] have shown that common knowledge cannot be gained, if it was not present initially, in a system which does not admit of simultaneous events. The following corollary to lemma 3 shows that common knowledge can *neither be gained nor lost* in distributed systems.

Corollary. *In a system with more than one process, for any predicate b, b is common knowledge is a constant.* □

Proof. For any process p, b *is common knowledge* $=p$ *knows* (b *is common knowledge*). Hence, b *is common knowledge* is local to every p. Applying lemma 3, b *is common knowledge* is a constant. □

It is possible to show that even weaker forms of knowledge cannot be gained or lost in our model of distributed systems. Process sets P, Q have *identical knowledge* of b means,

P *knows* $b=Q$ *knows* b

Corollary. *If P, Q are disjoint and have identical knowledge of b then P knows b (and also Q knows b) is a constant.* □

Proof. P *knows* b is local to P and Q *knows* b is local to Q. From P *knows* $b=Q$ *knows* b, they are also local to Q and P respectively. The result follows directly from lemma 3. □

Corollary. *If P, Q are disjoint and P sure $b=Q$ sure b, then P sure b (and also Q sure b) is a constant.* □

4.3 How knowledge is transferred

We show in this section that chains of knowledge are gained or lost in a sequential manner.

Theorem 4. *For arbitrary process sets $P_1 \ldots, P_n$, $n \geqq 1$, predicate b and computations x, y,*

(P_1 *knows* ... P_n *knows* b *at* x *and* $x[P_1 \ldots P_n]y$) *implies* (P_n *knows* b *at* y). □

Proof. Proof is by induction on n. For $n=1$, P_1 *knows* b *at* x, $x[P_1]y$ *implies* P_1 *knows* b *at* y, trivially.

Assume the induction hypothesis for some $n-1$, $n>1$, and assume

P_1 *knows* ... P_n *knows* b *at* x and $x[P_1 \ldots P_n]y$.

We shall prove P_n *knows* b *at* y.

From $x[P_1 \ldots P_n]y$, we conclude that there is a z such that,

$x[P_1 \ldots P_{n-1}]z$ and $z[P_n]y$.

From $x[P_1 \dots P_{n-1}]\, z$ and P_1 *knows* ... P_{n-1} *knows* (P_n *knows b*) *at x*, we conclude, using induction, P_{n-1} *knows* P_n *knows b at z*. Hence, P_n *knows b at z*.

Since $z\,[P_n]\,y$, P_n *knows b at y*. □

Corollary. *For arbitrary process sets* $P_1 \dots P_n$, $n \geqq 1$, *predicate b and computations x, y,*

(P_1 *knows* ... P_{n-1} *knows* $\sim P_n$ *knows b at x and* $x[P_1 \dots P_n]\, y$) *implies* $\sim P_n$ *knows b at y*. □

Note. For $n=1$ antecedant is, $\sim P_n$ *knows b at x*.

Corollary. *Theorem* 4 *holds with knows replaced by sure in* "P_n *knows*".

Theorem 4 can be applied to (1) $x \leqq y$ (knowledge is lost) and (2) $y \leqq x$ (knowledge is gained). Using theorem 1, we can deduce that there is a process chain $\langle P_1 \dots P_n \rangle$ in the former case and $\langle P_n \dots P_1 \rangle$ in the latter case. We first prove a simple lemma about the effect of receive or send on knowledge: we show that certain forms of knowledge cannot be lost by receiving nor gained by sending.

Lemma 4. *(How events at a process change its knowledge)*

Let b be a predicate which is local to $\bar{P}$ *and* $(x;e)$ *a computation where e is an event on P.*

1. *e is a receive:* {*knowledge is not lost*}
   (*P knows b at x*) *implies* (*P knows b at* $(x;e)$)
2. *e is a send:* {*knowledge is not gained*}
   (*P knows b at* $(x;e)$) *implies* (*P knows b at x*)
3. *e is an internal event:*
   {*knowledge is neither lost nor gained*}
   (*P knows b at x*)$=$(*P knows b at* $(x;e)$). □

Proof. We prove only (1). Consider any z such that $(x;e)[P]\,z$. We will show *b at z* and hence it follows that *P knows b at* $(x;e)$.

Since $z[\bar{P}]\,z$, we have $(x;e)[P\bar{P}]\,z$.

From theorem 3, since e is a receive, $x[P\bar{P}]$ z. Since b is local to $\bar{P}$,

P knows b $=$ *P knows* $\bar{P}$ *knows b*.

From theorem 4,

(*P knows* $\bar{P}$ *knows b at x*, $x[P\bar{P}]\,z$) *implies*
($\bar{P}$ *knows b at z*)

($\bar{P}$ *knows b at z*) *implies* (*b at z*).

This completes the proof. □

Corollary. (*b is local to* $\bar{P}$, $\sim P$ *knows b at x, P knows b at y,* $x \leqq y$) *implies* (*P receives a message in* (x, y)). □

Corollary. (*b is local to* $\bar{P}$, *P knows b at x,* $\sim P$ *knows b at y,* $x \leqq y$) *implies* (*P sends a message in* (x, y)). □

Theorem 5. *(How knowledge is gained) Let x, y be computations where* $x \leqq y$, $\sim$(P_n *knows b*) *at x and* (P_1 *knows* ... P_n *knows b*) *at y, for arbitrary process sets* $P_1 \dots P_n$, $n \geqq 1$. *Then there is a process chain* $\langle P_n \dots P_1 \rangle$ *in* (x, y). *Furthermore, if b is local to* $\bar{P}_n$ *then* P_n *has a receive event in* (x, y) *such that b at z holds for every prefix z of y which includes the corresponding send event.* □

Theorem 6. *(How knowledge is lost) Let x, y be computations where* $x \leqq y$, P_1 *knows* ... P_n *knows b at x and* $\sim P_n$ *knows b at y, for arbitrary process sets* $P_1 \dots P_n$, $n \geqq 1$. *Then there is a process chain* $\langle P_1 \dots P_n \rangle$ *in* (x, y). *Furthermore, if b is local to* $\bar{P}_n$ *then* P_n *has a send event in* (x, y). □

Observe that the statements of the two theorems are not entirely symmetric for receive and send events. The reason is that every computation including a receive must also include the corresponding send, but not conversely.

5 Applications of the results

We discuss a few applications of the theory developed so far in the paper.

5.1 When is a process unsure about a predicate?

We show that it is impossible for processes P to track the change in value of a local predicate of $\bar{P}$, at all times; P must be unsure about the value of this predicate while it is undergoing change.

Lemma 6. *(Interval of uncertainty:) Let b be a predicate local to* $\bar{P}$. *Let, b at* $x \neq b$ *at* $(x;e)$ *for some computation* $(x;e)$. *Then P unsure b at x and P unsure b at* $(x;e)$. □

Proof. Since b is local to $\bar{P}$ and its value changes as a result of event e, e is not on P. Therefore, $x[P](x;e)$ and hence *P knows b at* $x=$ *P knows b at* $(x;e)$. Since *b at* $x \neq b$ *at* $(x;e)$, both *P knows b at x* and *P knows b at* $(x;e)$ are *false*. Analogously, *P knows* $\sim b$ *at x* and *P knows*

$\sim b$ *at* $(x;e)$ are both *false*. This completes the proof. □

What does this lemma imply about the event e on $\bar{P}$ which changes the value of local predicate b of $\bar{P}$? It follows that P must be unsure about b for event e to occur. Furthermore, we show that if e is internal or send then a necessary condition for occurrence of e is that $\bar{P}$ *knows* P *unsure* b before application of e.

Theorem 7. *Let b be local to $\bar{P}$. For a computation $(x;e)$, where*

b at $x \neq b$ at $(x;e)$

($\bar{P}$ knows P unsure b) at x, if e is an internal or send event on $\bar{P}$,
($\bar{P}$ knows P unsure b) at $(x;e)$, if e is internal or receive on $\bar{P}$ □

Proof. Consider any y for which $x[\bar{P}]y$. From the principle of computation extension, $(y;e)$ is also a computation; hence $(x;e)[\bar{P}](y;e)$.

b is local to $\bar{P}$, hence: *b at $x = b$ at y*
and, b at $(x;e) = b$ at $(y;e)$.

From, *b at $x \neq b$ at $(x;e)$* it follows that : *b at $y \neq b$ at $(y;e)$.*
Hence, from lemma (6), *P unsure b at y.*
From the definition of knowledge, *$\bar{P}$ knows P unsure b at x.* The other part is similarly proven. □

Corollary. *Let b be local to $\bar{P}$. For a computation $(x;e)$, where e is an internal event on $\bar{P}$, if:*

b at $x \neq b$ at $(x;e)$

then for any $y, x \leqq y$, where $\bar{P}$ has no send event in (x,y):

$\bar{P}$ knows P unsure b at y. □

Proof. From Theorem 7, *$\bar{P}$ knows P unsure b at x.* Since $\bar{P}$ sends no message in (x,y), from Lemma 4, $\bar{P}$ can lose no knowledge and hence, *$\bar{P}$ knows P unsure b at y.* □

5.2 Detection of process failure is impossible

Traditional techniques for process failure detection based on time-outs assume certain execution speeds for processes and maximum delays for message transfer. It is generally accepted that detection of failure is impossible without using time-outs, a fact that we prove formally.

We model failure of $\bar{P}$ as follows. Let f be a local predicate of $\bar{P}$ denoting that $\bar{P}$ has failed. We assume that (1) f is initially *false*, and (2) $\bar{P}$ may fail at any time, i.e. for every x for which $\sim f(x)$, there is an internal event e on $\bar{P}$ such that $f(x;e)$ and (3) $\bar{P}$ sends no message as long as f holds. Under these constraints, we show that P is always *unsure* of failure of $\bar{P}$. In fact, we show that *$\bar{P}$ knows P unsure f* at all computations y. Note that we do not require failure to persist, i.e. it is entirely possible to have $x \leqq y, f(x)$ and $\sim f(y)$.

Theorem 8. *$\bar{P}$ knows P unsure f at y, for all y,* □

Proof. If $\sim f(y)$, there is an internal event e on $\bar{P}$ such that $f(y;e)$. From Theorem 7, *$\bar{P}$ knows P unsure f at y.* If $f(y)$, then from the fact that f is *false* initially, there is some $(x;e)$, $(x;e) \leqq y$, such that, $\sim f(x)$ and $f(x;e)$. Without loss in generality, we may assume that $\bar{P}$ stays failed after $(x;e)$ until y. Since e is an internal event and $\bar{P}$ stays failed after $(x;e)$, there is no send event on $\bar{P}$ in (x,y). Hence, from corollary to Theorem 7, *$\bar{P}$ knows P unsure f at y.* □

5.3 Mutual exclusion

Consider a system of processes in which every process p has a local predicate cs_p and for every pair of processes p, q and every computation x, $\sim(cs_p$ *and* $cs_q)$ *at* x. Intuitively, cs_p denotes that p is in its critical section and the restriction that no two processes can simultaneously be in their critical sections, is captured by the last requirement. We show that in every computation of a solution to the mutual exclusion problem (in our model), there is a process chain $\langle p_1 \dots p_n \rangle$, where p_i is the ith process to enter its critical section.

Theorem 9. *For any $x, y, x \leqq y$, cs_p at x and cs_q at y implies that there is a process chain $\langle pq \rangle$ in (x,y).* □

Proof. Observe that cs_p *implies* $\sim cs_q$, and $\sim cs_q$ *implies* (q *knows* $\sim cs_q$). Also, cs_q *implies* ($\sim q$ *knows* $\sim cs_q$). Hence, (cs_p *at* x) *implies* (p *knows* q *knows* $\sim cs_q$ *at* x) and (cs_q *at* y) *implies* ($\sim q$ *knows* $\sim cs_q$ *at* y). The result follows from theorem (6). □

We can show, based on the observation given below, that a solution to the distributed dining philosophers problem appearing in [1] requires no more than twice the number of messages in an optimal scheme. In the distributed dining philosophers problem, philosophers are placed at vertices of an undirected graph and one fork is placed on each edge. A philosopher requires forks on all incident edges to eat and hence neighboring philosophers cannot eat simultaneously.

Observation. For neighboring philosophers p,q, there is a process chain $\langle pq \rangle$ in (x,y) where p eats at x, q eats at y and $x \leqq y$. Hence at least one message must be sent by p to q between an eating session by p and a subsequent eating session by q. The solution in [1] employs two messages between an eating session by p and a subsequent eating session by q.

5.4 Complexity of termination detection

We show that any algorithm which detects termination of an underlying computation requires at least as many overhead messages, in general, for detection as there are messages in the underlying computation. We prove our result by considering a specific underlying computation.

Consider a system of two processes A, B in which messages may be sent from A to B and from B to A. Each process is initially in a *tossing* state. Each process in *tossing* state decides nondeterministically (by a coin toss, for instance) to enter either a *receiving* or a *sending* state. A process in the *receiving* state waits until it receives a message and then returns to the tossing state. A process in the sending state sends a message and then returns to the tossing state. If both processes are in the receiving state and every message sent has been received, then both processes will remain waiting forever. The goal of the termination detection algorithm is to detect such a situation.

In the sequel, we use *underlying computation* to mean the computation associated with coin tossing, sending and receiving of messages as described above. The termination detection algorithm superimposes an *overhead computation* on the underlying computation at each process; we use *computation* to mean the underlying computation and overhead computation together. *Overhead messages* and *underlying messages* belong to the corresponding computations.

The overhead computation at a process can observe the state of the underlying computation, but cannot affect it. The overhead computation may have its own associated states and it may send messages (to the overhead computation at the other process) even when the underlying computation is waiting to receive. However, a message is received only when the underlying computation is waiting to receive. We require that whenever the termination detection algorithm reports termination, the underlying computation has terminated (both processes are in receiving state and there is no underlying message in channels); furthermore, for every computation x in which the underlying computation has terminated, there is a computation $y, x \leqq y$, in which termination is reported by the overhead computation at one of the processes.

We show that for any $k, k \geqq 0$, there is a computation in which k underlying messages are sent and received and at least k overhead messages are sent. The plan of the proof is as follows. We first show that in order for termination to be detected, an overhead message is sent by some process, without its first receiving a message, after the underlying computation terminates; this fact is proven directly from the theorem of knowledge gain, because detecting termination amounts to gaining knowledge.

Next, we show that a process is sometimes required to send an overhead message even when the underlying computation has not terminated, because the computation may be isomorphic (with respect to this process) to a computation in which the underlying computation has terminated. Using these two results, we construct a computation, of the required type, for any $k, k \geqq 0$.

Theorem 10. *For any $k, k \geqq 0$, there is a computation in which k underlying messages are sent and received and at least k overhead messages are sent.* $\square$

Proofs. We will prove a slightly stronger result, $I(k)$, for any $k, k \geqq 0$, where $I(k)$ is: there is a computation in which k underlying messages are sent and received, at least k overhead messages are sent and both processes are in tossing state at the end of the computation. Proof is by induction on k.

For $k=0$: $I(0)$ holds for the *null* computation, from the initial condition.

Let x be a computation for which $I(k)$ holds for some $k, k \geqq 0$. We show a computation z in which $I(k+1)$ holds.

Let $tr_A(tr_B)$ denote an internal event at $A(B)$ whereby the process transits from tossing state to receiving state; similarly, let $ts_A(ts_B)$ denote the transition from the tossing to sending state.

Consider the computation $x' = (x; tr_A; tr_B)$. Since no underlying message is in transit in x and both processes are waiting to receive in x', x' has a terminated underlying computation.

For each process, "process is in receiving state" is a local predicate of the process. This predicate value, for each process, is *false* at x. If a process (say B) detects termination at some $y, x' \leqq y$, then B *knows* A is in receiving state *at* y. Therefore, B gains knowledge about A and, applying the knowledge gain theorem (theorem 5), there is a process chain $\langle A, B\rangle$ in (x', y). Therefore, in general, either there is a process chain $\langle A\, B\rangle$ or a process chain $\langle B\, A\rangle$ in (x, y). Let y' be such that $x \leqq y' < y$, (x, y') contains no process chain $\langle A\, B\rangle$ or $\langle B\, A\rangle$ and (x, y') contains a message send (which must be an overhead message) by some process, say A.

Let $w = (x; tr_A; ts_B$; B sends underlying message). Since there is no process chain $\langle A\, B\rangle$ or $\langle B\, A\rangle$ in (x, y') or (x, w), we can apply the fusion theorem (theorem 2) to y' and w to obtain a computation w', where $x \leqq w'$, $w'[B]\, w$ and $w'[A]\, y'$. In computation (x, w'), B has sent an underlying message and A has sent an overhead message before receiving the underlying message. To complete the proof, we note that there is an extension z of w' in which A receives the underlying message sent to it by B. Computation z satisfies $I(k+1)$. □

6 Discussion

We have shown that isomorphism between system computations with respect to a process is a useful concept in reasoning about distributed systems. Isomorphism forms the basis for defining and deriving properties about knowledge. "Scenarios" have been used [7] to show impossibility of solving certain problems; in our context, a scenario is a computation, and isomorphism is the formal treatment of equivalence between scenarios. Theorems on knowledge transfer provide lower bounds on numbers of messages required to solve certain problems. We have used isomorphism as the basis of fusion theorem and related isomorphism to semantics of send, receive and internal events.

In this paper, we have not defined processes in terms of their states. The notion of isomorphism between computations could be defined in terms of process states as follows: two computations x and y are *state-isomorphic* with respect to a process p means the state of p after x is the same as its state after y. Observe that x and y are *isomorphic* with respect to a process implies they are *state-isomorphic* with respect to that process. With knowledge defined in terms of state-isomorphism, a process may lose knowledge by an internal event, that is, by merely by changing its state. However, knowledge can be gained only be receiving messages. In other words, processes may "forget" on their own but cannot learn without receiving information. The theorem of knowledge transfer applies even with knowledge defined in terms of *state-isomorphism*. This is an area worth pursuing, as it may provide insight into designs of processes.

Our model does not have the notion of time. If there is a global clock common to all processes then processes may learn or forget merely by the passage of time. For instance, in time-division multiplexing, the mutual exclusion problem is solved by letting the i-th process be in its critical section during the i-th slot in the time cycle. In this case, a computation is a tuple consisting of the "current" time and a sequence of timed-events where each timed-event is a pair (time, event). The concept of isomorphism remains valid, though the knowledge transfer theorems no longer hold, because knowledge can be gained and lost merely by the passage of time.

It is tempting to define *belief* in terms of isomorphism as follows: process p *believes* b at x means b holds for most (in measure-theoretic terms) computations isomorphic to x with respect to p. Unfortunately, there do not appear to be clean results on the gain/loss of belief or belief transfer.

In this paper, when we say a process knows b, we allow b to be an arbitrary predicate; b may be temporal, for instance of the form: *eventually* b'. For example, in a commit protocol a process committing itself to a value v knows that all correct processes will eventually commit to v. Results about knowledge transfer-gain or loss-still hold.

Acknowledgement. We are indebted to Shmuel Katz, Joe Halpern, E.W. Dijkstra and Bengt Jonsson for their comments. Particular thanks go to Ernie Cohen for a careful reading of the manuscript and insightful comments.

References

1. Chandy KM, Misra J (1984) Drinking philosophers problem. TOPLAS, October 1984
2. Fischer MJ, Lynch N, Paterson M (1985) Impossibility of distributed consensus with one faulty process. J ACM, April 1985
3. Halpern JY, Moses Y (1984) Knowledge and common knowledge in a distributed environment. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1984
4. Hintikka J (1962) Knowledge and belief. Cornell University Press
5. Lamport L (1978) Time, Clocks and the orderings of events in a distributed system. Communications of the ACM 21: 558-564
6. Lehmann D (1984) Knowledge, common knowledge, and related puzzles. ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing, Vancouver, Canada, August 1984
7. Lynch N & Fischer M (1982) A lower bound for the time to assure interactive consistency. Information Proc Letters 14, 4, June 1982