USING CONCURRENT STATES TO REBUILD THE THEORY FOR DISTRIBUTED COMPUTING

Y.C. Tay and X.B. Shen
tay@acm.org shenxb@comp.nus.edu.sg
Department of Mathematics and Department of Computer Science
National University of Singapore

Abstract Much of the theory for distributed computing is constructed with global states, i.e. simultaneous local states across space. However, it is known — since the discovery of relativity — that global states do not exist. It is therefore inevitable that, to progress as a science, the theory must be rebuilt without using global states. This paper suggests that the theory should be based on concurrent states.

Keywords: Distributed computing, time, concurrent states.

1 Introduction

TODC, or the Theory of Distributed Computing (mutual exclusion, deadlock detection, leader election, Byzantine agreement, self-stabilization, etc.) is largely constructed with global states, i.e. vectors of the form $(s_1(t), \ldots, s_n(t))$ where t is real time (or global clock, Nature's clock, etc.) and $s_i(t)$ is the state of process i at time t. In other words, $s_1(t), \ldots, s_n(t)$ are simultaneous states of process $1, \ldots, n$. Yet, it is well-known that real time (or Newton's universal time) is a fiction [1, 2], that there is no simultaneity across space, so global states are not only unobservable, they simply do not exist. Another popular model that uses a total, global ordering of events has the same weakness. It follows that much of TODC has a shaky foundation.

One could argue that, when constructing a theory in computer science, we must always pick appropriate abstractions (messages, events, states, etc.) that approximate reality, and the global state is one such abstraction; after all, Newtonian mechanics is a very good approximation of reality. However, whereas a theory in physics focuses on deriving descriptions and predictions that approximate nature, a theory like TODC is mostly about correctness proofs, and correctness has no approximation: A protocol for achieving approximate agreement [3], say, is either correct, or not.

We believe that, eventually, TODC will be overhauled to take into account the dependence between space and time. And this is likely to happen sooner, rather than later: Already, quantum mechanics has made inroads into the theories of computability, complexity and cryptography, relativistic effects have been observed in Global Positioning Systems [4], and the design of an interplanetary Internet has begun [5]. A theory meant for a technology that prides itself on its breath-taking pace must surely incorporate, soon, the effects and implications of a scientific discovery that is almost 100 years old.

Relativistic effects (like time dilation) have interesting ramifications on specific problems in TODC (like clock synchronization), but it is the implication that there is no global state that has pervasive impact on TODC.

Since every process decides its actions based only on local observations, a correctness proof should (if not for scientific reasons, then for aesthetics) also be based on local observations, rather than on a global state. However, the point of distributed computing is to achieve some global objective; for example, in distributed consensus, the objective is to have all correct processes agree on a common (i.e. global) value despite failures. Can one prove that a global objective is (or cannot be) achieved without using global states in the proof? For instance, to prove that a deadlock detector is correct, one must start with the definition of a deadlock, which — intuitively — is a set of processes waiting for each other *simultaneously*. Can deadlocks be defined without using global states?

This paper introduces the notion of a *concurrent state* as a generalization of the global state, and as a basis for rebuilding TODC. (We are encouraged by the fact that Dirac had also tried to replace the non-relativistic concept of "the same instant of time" in quantum field theory by a "many-time formalism" that has a time associated with each particle [6, p54, p261].) We demonstrate its relevance to Chandy and Lamport's snapshot protocol [7], its application in Tay and Loke's theory for deadlocks [8], and its use for proving the Fischer-Lynch-Paterson impossibility theorem in distributed consensus [9].

2 Model: Time, State, Events and Transition

A distributed system consists of processes. Let P be a process. We assume there is a non-empty set of *P-times* (or *local times*) $\mathtt{T}_P$. A P-time is a logical entity; it may be a value on a physical clock that P can read, or an element in the total order of events in a P that migrates from site to site. The elements of $\mathtt{T} = \bigcup_P \mathtt{T}_P$ are called *times*.

Each process intersperses its computations with message transmissions to, and reception from, another process. The nature of the distributed system (client/server, shared-memory or massively parallel architecture, process migration, holographic routing, etc.) is entirely contained in the messages, which are an abstraction for communication in the system and therefore logical in nature, rather than physical. For instance, a message may be implemented as a procedure call or an access to shared memory (cf. Bar-Noy and Dolev's translations between message-passing and shared-memory [10] and Valiant's messages in his bridging model [11]).

A message is represented by a predicate $\mathrm{msg}(P_{\mathrm{send}}, P_{\mathrm{recv}}, t_{\mathrm{send}}, t_{\mathrm{recv}})$, which is true if and only if a message was sent by process P_{send} at local time $t_{\mathrm{send}} (\in \mathtt{T}_{P_{\mathrm{send}}})$ and received by process P_{recv} at local time $t_{\mathrm{recv}} (\in \mathtt{T}_{P_{\mathrm{recv}}})$. We assume the local times in $\mathtt{T}_P$ are totally ordered by some $\leq_P$. We can therefore represent $<_{P_{\mathrm{send}}}$ and $<_{P_{\mathrm{recv}}}$ as lines, and

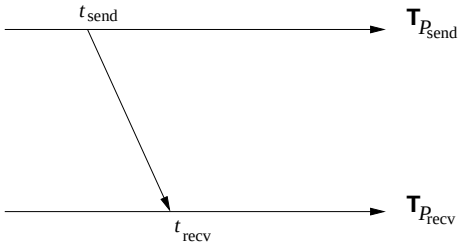

Figure 1 $\text{msg}(P_{\text{send}}, P_{\text{recv}}, t_{\text{send}}, t_{\text{recv}})$

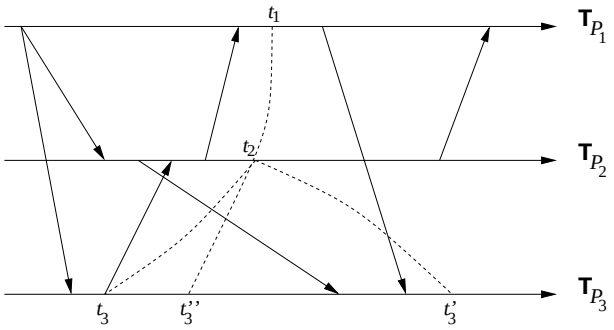

Figure 2 Examples of cuts.

$\text{msg}(P_{\text{send}}, P_{\text{recv}}, t_{\text{send}}, t_{\text{recv}})$ by an arrow between them, as in Figure 1.

$\leq_P$ and messages induce a partial ordering $\preceq$ on all the times in $\mathtt{T}$, as follows [12]: For all $t_1, t_2 \in \mathtt{T}$, $t_1 \prec t_2$ if and only if (i) $t_1 <_P t_2$ for some P, where t_1 and t_2 are P-times, (ii) $\text{msg}(P_{\text{send}}, P_{\text{recv}}, t_1, t_2)$ for some P_{send} and P_{recv}, or (iii) $t_1 \prec t$ and $t \prec t_2$ for some $t \in \mathtt{T}$. We assume that the laws of physics guarantee the existence of such an ordering. In particular, for P-times t_1 and t_2, $t_1 \prec t_2$ if and only if $t_1 <_P t_2$. We write $t_1 < t_2$ if $t_1 <_P t_2$ and P is clear from the context. A system may be running multiple protocols, in which case they induce their own (albeit compatible) partial orders.

Consider a system with n processes $P_1, \ldots, P_n$. A (time) *cut* is a vector of local times $(t_1, \ldots, t_n)$, where $t_i \in \mathtt{T}_{P_i}$ for all i. In Figure 2, (t_1, t_2, t_3), (t_1, t_2, t_3') and (t_1, t_2, t_3'') are examples of cuts. We say $(t_1, \ldots, t_n) \leq (t_1', \ldots, t_n')$ if and only if $t_i \leq t_i'$ for all i.

Two times t and t' are *concurrent*, denoted $t \| t'$, if and only if $t \not\prec t'$ and $t' \not\prec t$. In Figure 2, $t_1 \| t_3''$ but $t_1 \not\| t_3$ and $t_1 \not\| t_3'$. Since $\leq_P$ is a total ordering, we have $t \not\prec t$, so $t \| t$ for any P-time t. A cut $(t_1, \ldots, t_n)$ is *concurrent* if and only if $t_i \| t_j$ for all i and j. In Figure 2, (t_1, t_2, t_3'') is concurrent, but (t_1, t_2, t_3) and (t_1, t_2, t_3') are not. In a global time model, $\mathtt{T}_P = \mathtt{T}$ for all processes, and only cuts of the form $(t, \ldots, t)$ are concurrent.

We are interested in the progress of a system in which the processes $P_1, \ldots, P_n$ run a protocol $\mathcal{X}$; i.e. each P_i executes an algorithm $\mathcal{X}$ that includes exchange of messages with other processes.

Changes at a process P_i are modeled by changes in its *state* at time t_i, denoted $S_i(t_i)$. Depending on the context, $S_i(t_i)$ may include the process's memory contents, the messages it has sent and received, the local time, its location (in the case of migrating processes), etc. Different problems in distributed computing focus on different constituents of $S_i(t_i)$. For example, the snapshot protocol records the part of $S_i(t_i)$ that is affected by some *underlying computation*, distributed consensus considers only the memory contents, and clock synchronization studies adjustments of t_i. We use $S_i(t_i).\mathit{part}$ to denote some particular part of $S_i(t_i)$; e.g. $S_i(t_i).\mathit{memory}$ refers to the memory contents, and $S_i(t_i).\mathtt{T} = t_i$. $S_i(t_i)$ is changed by execution at P_i.

There are two conflicting considerations when modeling *executions*: We want to discretize an execution of a process into a sequence of *atomic* events, by which we mean an event cannot be interrupted, except by a failure, and can be assigned a time instant t so we can say the event occurs at time t. On the other hand, an event changes the state of a process, and it will be awkward to associate more than one state to t. We resolve this conflict with the following model:

- every event e_i of a process P_i *occurs* at some P_i-time t_i;
- e_i spans a time interval from t_i to t_i+, where $t_i < t_i+$;
- if P_i has another non-failure event $e_i{}'$ occurring at $t_i{}'$, then $t_i{}' < t_i$ or $t_i+ < t_i{}'$;
- if e_i changes the state of P_i from $S_i{}'$ to $S_i{}''$, then $S_i{}'$ is the state at t_i and $S_i{}''$ the state at t_i+, denoted $S_i{}' = S_i(t_i)$ and $S_i{}'' = S_i(t_i+)$.

If $\mathbf{t} = (t_1, \ldots, t_n)$ is a cut and s_i is some part of S_i, we call $\mathbf{s}(\mathbf{t}) = (s_1(t_1) \ldots, s_n(t_n))$ a *state-cut*. If $\mathbf{t}$ is a concurrent cut, we call $\mathbf{s}(\mathbf{t})$ a *concurrent state*. Some work in the literature [13] are based on consistent cuts, where $(t_1, \ldots, t_n)$ is *consistent* if and only if there is no $\mathrm{msg}(P_i, P_j, t_{\mathrm{send}}, t_{\mathrm{recv}})$ such that $t_i < t_{\mathrm{send}}$ and $t_{\mathrm{recv}} < t_j$. In Figure 2, (t_1, t_2, t_3) and $(t_1, t_2, t_3{}'')$ are consistent but $(t_1, t_2, t_3{}')$ is not. Consistent is weaker than concurrent: Suppose $(t_1, \ldots, t_n)$ is not consistent, so there are t_i and t_j such that $\mathrm{msg}(P_i, P_j, t_{\mathrm{send}}, t_{\mathrm{recv}})$, $t_i < t_{\mathrm{send}}$ and $t_{\mathrm{recv}} < t_j$. Then $t_i \prec t_j$ so $(t_1, \ldots, t_n)$ is not concurrent.

Transitions in our model are driven by the following lemma:

One-Step Lemma
Suppose $\mathbf{t} = (t_1, \ldots, t_n)$, $\mathbf{t}$ is a concurrent cut, P_i has an event e_i at t_{e_i}, where $t_i < t_{e_i}$, and there is no event at P_i in the time interval (t_i, t_{e_i}). Let $\mathbf{t}' = (t_1{}', \ldots, t_n{}')$ where $t_i{}' = t_{e_i}+$ and $t_k{}' = t_k$ for $k \neq i$. If e_i is not a message reception, or if we have $\mathrm{msg}(P_j, P_i, t_{\mathrm{send}}, t_{e_i})$ and $t_{\mathrm{send}} < t_j$, then $\mathbf{t}'$ is a concurrent cut.

Proof
Consider $k \neq i$. If $h \neq i$, then $t_k{}' = t_k$, $t_h{}' = t_h$, so $t_k{}' \| t_h{}'$ since $t_k \| t_h$. Next, if $t_i{}' \prec t_k{}'$, then $t_i < t_{e_i} < t_{e_i}+ = t_i{}' \prec t_k{}' = t_k$, contradicting $t_i \| t_k$. Now suppose $t_k{}' \prec t_i{}'$, i.e. $t_k \prec t_{e_i}+$. Since $t_k \| t_i$, $t_k \prec t_{e_i}+$ must be induced by some $\mathrm{msg}(P_j, P_i, t_{\mathrm{send}}, t_{\mathrm{recv}})$ where $t_i < t_{\mathrm{recv}} < t_{e_i}+$ and $t_k \preceq t_{\mathrm{send}}$. P_i has no events in the interval $(t_i, t_{e_i}+)$ except at t_{e_i}, so $t_{\mathrm{recv}} = t_{e_i}$ and $t_{\mathrm{send}} < t_j$. Thus $t_k \preceq t_{\mathrm{send}} < t_j$, contradicting $t_k \| t_j$. □

Let $\mathbf{t}$, $\mathbf{t}'$ and e_i be as in the One-Step Lemma. We denote their relationship by $\mathbf{s}(\mathbf{t}) \vdash^{e_i} \mathbf{s}(\mathbf{t}')$. For two concurrent cuts $\mathbf{t}$ and $\mathbf{t}''$, $\mathbf{t} \leq \mathbf{t}''$, we write $\mathbf{s}(\mathbf{t}) \vdash^{*} \mathbf{s}(\mathbf{t}'')$ if and only if $\mathbf{s}(\mathbf{t}) = \mathbf{s}(\mathbf{t}'')$ or $\mathbf{s}(\mathbf{t}) \vdash^{e_i} \mathbf{s}(\mathbf{t}') \vdash^{*} \mathbf{s}(\mathbf{t}'')$ for some event e_i and concurrent cut $\mathbf{t}' \leq \mathbf{t}''$. Two events at different processes commute in the following sense:

Commutativity Lemma
Suppose $\mathbf{t}$ is a concurrent cut, $\mathbf{s}(\mathbf{t}) \vdash^{e_i} \mathbf{s}(\mathbf{t}^{e_i})$ and $\mathbf{s}(\mathbf{t}) \vdash^{e_j} \mathbf{s}(\mathbf{t}^{e_j})$, where events e_i and e_j occur at some processes P_i and P_j, $i \neq j$. Define $\mathbf{t}' = (t_1{}', \ldots, t_n{}')$ where $t_i{}' = t_i^{e_i}$,

$t_j' = t_j^{e_j}$ and $t_k' = t_k$ for $k \neq i, j$. Then $\mathbf{t}'$ is a concurrent cut and $\mathbf{s}(\mathbf{t}) \vdash^{e_i} \mathbf{s}(\mathbf{t}^{e_i}) \vdash^{e_j} \mathbf{s}(\mathbf{t}')$ and $\mathbf{s}(\mathbf{t}) \vdash^{e_j} \mathbf{s}(\mathbf{t}^{e_j}) \vdash^{e_i} \mathbf{s}(\mathbf{t}')$.

Proof

By the One-Step Lemma, $\mathbf{t}^{e_i}$ is a concurrent cut. Suppose we have $\mathrm{msg}(P_h, P_j, t_{\mathrm{send}}, t_j)$, where $t_{\mathrm{send}} < t_h$. Then $t_{\mathrm{send}} < t_i < t_i^{e_i}$ if $h = i$ and $t_{\mathrm{send}} < t_h = t_h^{e_i}$ if $h \neq i$, so $\mathbf{s}(\mathbf{t}^{e_i}) \vdash^{e_j} \mathbf{s}(\mathbf{t}')$ by the One-Step Lemma. Similarly, $\mathbf{s}(\mathbf{t}^{e_j}) \vdash^{e_i} \mathbf{s}(\mathbf{t}')$. □

3 Applications

We now apply the model to some problems in distributed computing.

3.1 Mutual Exclusion

One classical problem in distributed computing is *mutual exclusion* [14], in which the objective is to ensure that at most one process is privileged at any global time. We can use concurrent states to formulate this global objective, i.e. without using global time.

Suppose there is a predicate Q on local states, and s_i is *privileged* if and only if $Q(s_i)$ is true. Then the objective in the *Mutual Exclusion* problem is for a concurrent state $(s_i(t_i), \ldots, s_n(t_n))$ to satisfy the predicate $\mathcal{L}_{\mathrm{ME}}$ where at most one $s_i(t_i)$ is privileged, i.e.

$$\mathcal{L}_{\mathrm{ME}}(s_1(t_1), \ldots, s_n(t_n)) \equiv (\forall i \forall j \; t_i \| t_j) \rightarrow \; \forall i \forall j \neg((i \neq j) \wedge Q(s_i(t_i)) \wedge Q(s_j(t_j))). \qquad \square$$

3.2 Snapshot Protocol

The snapshot protocol is a general protocol that any system can run to record the local states of some underlying computation. Babaoğlu and Marzullo observed that the snapshot protocol records some $\mathbf{s}(\mathbf{t})$ such that $\mathbf{t}$ is consistent, but this $\mathbf{t}$ is not any arbitrary consistent cut. We now see why:

Proposition 1

The local states recorded in the snapshot protocol constitute a concurrent state. □

For their system, Chandy and Lamport made two assumptions: messages are pipelined, and the receipt of a marker (a special message used by the protocol) has no effect on the underlying computation. This second assumption implies that the recorded state is concurrent.

3.3 Distributed Deadlocks

Tay and Loke have developed an axiomatic theory for deadlocks. Since a deadlock is, intuitively, a set of processes waiting for each other simultaneously, the challenge lies

in not using global states in the theory. This is achieved by having local states that, in principle, record when a process sends a request for a resource ($\vec{P}R$), when the resource manager receives that request ($P\vec{R}$), when it grants the request ($\vec{R}P$), when a process receives notification of that grant ($R\vec{P}$), when it releases the resource ($\neg R\vec{P}$), when the manager receives the release message ($\neg\vec{R}P$), and when a process aborts (all times local).

If concurrent state is to be a generalization of global state, there should be some necessary and sufficient conditions for a deadlock that are in terms of concurrent states. Indeed, one can prove the following:

Proposition 2
Suppose $\mathbf{s(t)}$ is a concurrent state and in $s_i(t_i)$, for $i = 1, \ldots, n$, process P_i is holding resource R_{i+1} and waiting for resource R_i (i.e. $R_{i+1}\vec{P}_i$ and $\vec{P}_i R_i$, where $n+1 \equiv 1$). Then $P_1, \ldots, P_n$ are deadlocked at $t_1, \ldots, t_n$ over $R_1, \ldots, R_n$. □

(In the above statement, "deadlock" is as defined by Tay and Loke). A necessary condition is harder to state — the processes in some $\mathbf{s(t)}$ may be deadlocked without $\mathbf{t}$ being a concurrent cut — unless we make some extra assumptions (e.g. a process that is waiting for a resource may not send any messages.)

3.4 Distributed Consensus

A fundamental result in distributed computing is the FLP Theorem that says it is impossible for an asynchronous system to guarantee consensus despite one failure. We now apply our model to this result.

The part of local state that is of interest to the problem is the memory, so henceforth, $s_i(t_i) = S_i(t_i).memory$. The occurrence of events is determined by this $s_i(t_i)$. A process can *fail* at any time — this failure event f can occur even during another event. If P_i fails at t_i, then $s_i(t_i') = s_i^{fail}$ for all $t_i' > t_i$.

There are two distinguished registers named *input* and *output*; input values belong to $\{0, 1\}$ and output values belong to $\{\perp, 0, 1\}$. A process P_i begins executing the consensus protocol with an *initial state* s_i^{init}; the input in s_i^{init} is initialized by P_i or some other process, and the rest of s_i^{init} is specified by the protocol — in particular, the initial output value is $\perp$. P_i is *undecided* in $\mathbf{s(t)}$ if and only if $s_i(t_i)$ has output value $\perp$. For β=0,1, we say process P_i decides on β in concurrent state $\mathbf{s(t)}$ if and only if $s_i(t_i)$ has output value β. P_i cannot change its decision — if $s_i(t_i)$ has output value β, then $s_i(t_i')$ has the same value for all $t_i' > t_i$.

Let $\mathbf{s}^{init}(\mathbf{t}) = (s_1^{init}, \ldots, s_n^{init})$, where $\mathbf{t}$ is a concurrent cut and the inputs initialized in some way. The problem requires that the consensus protocol must satisfy the following requirements:

(C1) Consensus: Suppose $\mathbf{s}^{init}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$.

(a) If there are no failures, then there is $\mathbf{s}(\mathbf{t}'')$ such that $\mathbf{s}(\mathbf{t}') \vdash^* \mathbf{s}(\mathbf{t}'')$, with no failures, and some process decides in $\mathbf{s}(\mathbf{t}'')$.

(b) If processes P_i and P_j decide in $\mathbf{s}(\mathbf{t}')$, then they decide on the same value.

(C2) Nontriviality: There is some $\mathbf{s}^{init}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$, with no failures, where some P_i decides on 0 in $\mathbf{s}(\mathbf{t}')$; there is some $\mathbf{s}^{init}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}'')$, with no failures, where some P_j decides on 1 in $\mathbf{s}(\mathbf{t}'')$.

(C3) Fault-tolerance:

(a) Suppose there is one failure in $\mathbf{s}^{init}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$. Then there is $\mathbf{s}(\mathbf{t}') \vdash^* \mathbf{s}(\mathbf{t}'')$, with no failures, such that some process decides in $\mathbf{s}(\mathbf{t}'')$.

(b) There is no infinite sequence $\mathbf{s}^{init}(\mathbf{t}) \vdash^{e^{(1)}} \mathbf{s}(\mathbf{t}^{(1)}) \vdash^{e^{(2)}} \cdots$ in which at most one $e^{(\ell)}$ is a failure, and all processes are undecided in every $\mathbf{s}(\mathbf{t}^{(k)})$.

(C1a) and (C3a) say the protocol can always proceed to a decision, as long as there is no, or one, failure. (C1b) captures the meaning of consensus. (C2) ensures the protocol is nontrivial (e.g. does not make the same decision regardless of initialization), while (C3b) ensures that the protocol does not postpone a decision indefinitely. We can now use concurrent cuts to adapt Bridgland and Watro's proof of the impossibility result [15]. Due to space constraint, we just outline our proof with three lemmas.

To begin, $\mathbf{s}(\mathbf{t})$ is *bivalent* if and only if there are $\mathbf{s}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}^0)$ and $\mathbf{s}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}^1)$ where some P_i decides on 0 in $\mathbf{s}(\mathbf{t}^0)$ and some P_j decides on 1 in $\mathbf{s}(\mathbf{t}^1)$.

Bivalence Lemma
Assume the consensus protocol satisfies (C1). Then every process is undecided in a bivalent $\mathbf{s}(\mathbf{t})$. □

Next, for $\beta \in \{0, 1\}$, we say $\mathbf{s}(\mathbf{t})$ is *β-valent* if and only if, for every $\mathbf{s}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$, if P_i decides in $\mathbf{s}(\mathbf{t}')$, it decides on β.

Univalence Lemma
Assume the consensus protocol satisfies (C1).

(a) Let $\beta \in \{0, 1\}$. If $\mathbf{s}(\mathbf{t})$ is β-valent and $\mathbf{s}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$, then $\mathbf{s}(\mathbf{t}')$ is β-valent.

(b) If $\mathbf{s}^{init}(\mathbf{t}) \vdash^* \mathbf{s}(\mathbf{t}')$, then $\mathbf{s}(\mathbf{t}')$ cannot be both 0-valent and 1-valent. □

We may therefore call $\mathbf{s}(\mathbf{t})$ *univalent* if it is 0-valent or 1-valent.

Initialization Lemma
A consensus protocol that satisfies (C1), (C2) and (C3) must have a bivalent initialization $(s_1^{init}, \ldots, s_n^{init})$. □

Proposition 3 (FLP Theorem)
There is no consensus protocol satisfying (C1), (C2) and (C3).

References

1. P.C.W. Davies. *About Time: Einstein's Unfinished Revolution.* Simon & Schuster, New York, USA (1995).
2. V. Pratt. Modeling concurrency with partial orders. Int. J. Parallel Programming 15, 1(1986), 33–71.
3. D. Dolev, N.A. Lynch, S.S. Pinter, E.W. Stark and W.E. Weihl. Reaching approximate agreement in the presence of faults. J. ACM 33, 3(July 1986), 499–516.
4. G. Strang and K. Borre. *Linear Algebra, Geodesy, and GPS.* Wellesley-Cambridge Press (1997).
5. Interplanetary Internet Research Group Charter, `http://www.irtf.org/charters/ipnrg.html`.
6. S.S. Schweber. *QED And The Men Who Made It: Dyson, Feyman, Schwinger And Tomonaga.* Princeton University Press, Princeton, USA (1994).
7. K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. ACM Tans. Computer Systems 3, 1(Feb. 1985), 63–75.
8. Y.C. Tay and W.T. Loke. On deadlocks of exclusive AND-requests for resources. Distributed Computing 9 (May 1995), 77–94.
9. M.J. Fischer, N.A. Lynch and M.S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2(Apr. 1985), 374–382.
10. A. Bar-Noy and D. Dolev. Shared memory versus message-passing in an asynchronous distributed environment. Proc. ACM Symp. Principles of Distributed Computing, Edmonton, Canada (Aug. 1989), 307–318.
11. L. G. Valiant. A bridging model for parallel computation. Comm. of the ACM 33, 8(Aug. 1990), 103–111.
12. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Comm. ACM 21, 7(Nov. 1978), 558–565.
13. Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems. In *Distributed Systems*, 55–96, S. Mullender (ed.), Addison-Wesley, New York (1993).
14. E.W. Dijkstra. Solution of a problem in concurrent programming control, Comm. ACM 8, 9(Sept. 1965), 569.
15. M.F. Bridgland and R.J. Watro. Fault-tolerant decision making in totally asynchronous distributed systems, Proc. ACM Symp. Principles of Distributed Computing, Vancouver, Canada (Aug. 1987), 52–63.