*Appeared in Proceedings SSS'08 (Detroit, November 2008), Springer LNCS 5340, p. 246–262 (Best Paper Award)* November 26, 2008

# The Asynchronous Bounded-Cycle Model

Peter Robinson* and Ulrich Schmid

Technische Universität Wien
Embedded Computing Systems Group (E182/2)
Treitlstrasse 1-3, A-1040 Vienna (Austria)
{robinson,s}@ecs.tuwien.ac.at

**Abstract.** This paper shows how synchrony conditions can be added to the purely asynchronous model in a way that avoids any reference to message delays and computing step times, as well as any global constraints on communication patterns and network topology. Our Asynchronous Bounded-Cycle (ABC) model just bounds the ratio of the number of forward- and backward-oriented messages in certain ("relevant") cycles in the space-time diagram of an asynchronous execution. We show that clock synchronization and lock-step rounds can easily be implemented and proved correct in the ABC model, even in the presence of Byzantine failures. We also prove that any algorithm working correctly in the partially synchronous $\Theta$-Model also works correctly in the ABC model. Finally, we relate our model to the classic partially synchronous model, and discuss aspects of its applicability in real systems.

**Key words:** Fault-tolerant distributed algorithms, partially synchronous models, clock synchronization.

## 1 Introduction

Adding synchrony conditions, relating the occurrence times of certain events in a distributed system to each other, is the "classic" approach for circumventing impossibility results like [9] in fault-tolerant distributed computing. The following models in between synchrony and asynchrony, which are all sufficiently strong for solving the pivotal consensus problem, have been proposed in literature: (1) The Archimedean model [20] bounds the ratio between maximum end-to-end delays and minimal computing step times. (2) The classic partially synchronous models [5, 7] and the semi-synchronous models [3, 18] bound message delays as well as the ratio of minimal and maximal computing step times. (3) The $\Theta$-Model [4, 21, 22] bounds the ratio between the maximal and minimal end-to-end delay of messages simultaneously in transit. (4) The FAR-Model [8] assumes lower bounded computing step times and message delays with finite average. (5)

* This research is supported by the Austrian Science Foundation (FWF) projects P17757 and P20529. A brief announcement appeared in the proceedings of PODC'08.

The Weak Timely Link (WTL) models of [1, 11, 13] assume that only messages sent via certain links have bounded end-to-end delay. All these models refer to individual message delays and/or computing step times, and most of them involve explicit time bounds and system-wide (global) constraints. Somewhat an exception is (6) the MMR model [17] suggested for implementing failure detectors in systems with process crashes, which assumes certain order properties for round-trip responses.

This paper shows how to add synchrony assumptions—sufficiently strong for implementing lock-step rounds, and hence for solving any important distributed computing problem—to the asynchronous model in a way that entirely avoids (1) any reference to message delays and computing step times, and (2) any global constraint on communication patterns and network topology. More specifically, our *Asynchronous Bounded-Cycle* (ABC) model bounds the ratio of the *number* of forward- and backward-oriented messages in certain cycles ("relevant cycles") in the space-time diagram of an asynchronous execution. In fact, there is only one scenario that is admissible in the purely asynchronous model but not in the ABC model: A chain $C_1$ of $k_1$ consecutive messages, starting at process $q$ and ending at $p$, that properly "spans" (i.e., covers w.r.t. real-time, see Fig. 2) another causal chain $C_2$ from $q$ to $p$ involving $k_2 \geqslant k_1\Xi$ messages, for some model parameter $\Xi > 1$.

Consequently, individual message delays can be arbitrary, ranging from 0 to any finite value; they may even continuously increase. There is no relation at all between computing step times and/or message delays at processes that do not exchange messages; this also includes purely one-way communication ("isolated chains"). For processes that do exchange messages, message delays and step times in non-relevant cycles and isolated chains can also be arbitrary. Only *cumulative* delays of chains $C_1$ and $C_2$ in *relevant* cycles must yield the event order as shown in Fig. 2. That is, the *sum* of the message delays along $C_2$ must not become so small that $C_1$ could span $k_1\Xi$ or more messages in $C_2$. ABC algorithms can exploit the fact that this property allows to "time out" relevant message chains, and hence failure detection.

## 2 The ABC Model

We consider a system of $n$ distributed processes, connected by a (not necessarily fully-connected) point-to-point network with finite but unbounded message delays. We neither assume FIFO communication channels nor an authentication service, but we do assume that processes know the sender of a received message.

Every process executes an instance of a distributed algorithm and is modeled as a state machine. Its local execution consists of a sequence of atomic, zero-time computing steps, each involving the reception of exactly one[1] message, a state transition, and the sending of zero or more messages to a subset of the processes in the system. Since the ABC model is entirely time-free, i.e., does not

[1] An algorithm cannot learn anything from receiving multiple asynchronous messages at the same time, cp. [6].

introduce any time-related bounds, we restrict our attention to message-driven algorithms [4, 14]: Computing steps at process $p$ are exclusively triggered by a single incoming message at $p$, with an external "wake-up message" initiating $p$'s very first computing step; we assume that this very first step occurs before any message from another process is received.

Among the $n$ processes, at most $f$ may be Byzantine faulty. A faulty process may deviate arbitrarily from the behavior of correct processes as described above; it may of course just crash as well, in which case it possibly fails to complete some computing step and does not take further steps. In order to properly capture the interaction of correct and faulty processes, we conceptually distinguish the receive event that triggers a computing step and the computing step itself. If the process is correct, both occur at the same time. In case of a faulty receiver process, however, we separate the reception of a message, which is not under the receiver's control but initiated by the network, from the processing of this message, which is under the receiver's control and hence arbitrary in case of a faulty receiver. Consequently, even crashed processes eventually receive messages sent by correct processes, and since processes can only receive one message per step, there is a total order on the receive events at every process.

We can now specify admissible executions for our asynchronous message-driven system, cp. [4]: (1) If an infinite number of messages are sent to a correct process, it must execute infinitely many computing steps, and (2) every message sent by a correct process must be received by every [correct or faulty] recipient within finite time. Note that we do not say anything about messages sent by faulty processes here, which are usually unconstrained anyway.

The ABC model just puts one additional constraint on admissible executions. It is based on the space-time diagram [12], which captures the causal flow of information in an admissible execution $\alpha$. In order to properly include faulty processes, we just drop every message sent by a faulty process (along with both its send and receive step) in the space-time diagram. Note that a similar message dropping can be used for exempting certain messages, say, of some specific type or sent/received by some specific processes, from the ABC synchrony condition. After all, it is the particular algorithm that determines whether the order of certain receive events matters, i.e., whether the involved cycle is relevant or not.

**Definition 1 (Execution Graph).** *The* execution graph $G_\alpha$ *is the digraph corresponding to the space-time diagram of an admissible execution $\alpha$, with nodes $V(G_\alpha) = \Phi$ corresponding to the receive events in $\alpha$, and edges reflecting the happens-before relation [12] without its transitive closure: $(\phi_i, \phi_j)$ is in the edge relation $\rightarrow_\alpha \colon \Phi \times \Phi$ if and only if one of the following two conditions holds:*

1. *The receive event $\phi_i$ triggers a computing step where a message $m$ is sent from a correct process $p$ to a process $q$; event $\phi_j$ is the receive event of $m$ at $q$. We call the edge $\phi_i \rightarrow_\alpha \phi_j$* non-local edge *or simply* message *in $G_\alpha$.*
2. *The events $\phi_i$ and $\phi_j$ both take place at the same processor $p$ and there exists no event $\phi_k$ in $\alpha$ occurring at $p$ with $i < k < j$. The edge $\phi_i \rightarrow_\alpha \phi_j$ is said to be a* local edge.

We will simply write $G$ and $\rightarrow$ instead of $G_\alpha$ and $\rightarrow_\alpha$ when $\alpha$ is clear from the context. Note that we will also consider execution graphs of finite prefixes of executions.

A *causal chain* $\phi_1 \rightarrow \cdots \rightarrow \phi_l$ is a directed path in the execution graph, which consists of messages and local edges. The *length of a causal chain* $D$ is the number of non-local edges (i.e., messages) in $D$, denoted by $|D|$. A *cycle* $Z$ in $G$ is a subgraph of $G$ that corresponds to a cycle in the undirected shadow graph $\bar{G}$ of $G$.[2] Since messages cannot be sent backwards in time, every cycle can be decomposed into at least 2 causal chains having opposite directions. We now take a closer look at such cycles, which capture all causal information relevant for ABC algorithms.

**Definition 2 (Relevant Cycles).** *Let $Z$ be a cycle in the execution graph, and partition the edges of $Z$ into the* backward *edges $\hat{Z}^-$ and the* forward *edges $\hat{Z}^+$ as follows: Identically directed edges are in the same class, and*

$$|Z^+| \leqslant |Z^-|, \tag{1}$$

*where $Z^- \subseteq \hat{Z}^-$ and $Z^+ \subseteq \hat{Z}^+$ are the restrictions of $\hat{Z}^-$ resp. $\hat{Z}^+$ to non-local edges (messages). The* orientation *of the cycle $Z$ is the direction of the forward edges $Z^+$, and $Z$ is said to be* relevant *if all local edges are backward edges, i.e., if $\hat{Z}^+ = Z^+$; otherwise it is called* non-relevant.

Fig. 2 shows an example of a relevant cycle: Its orientation is opposite to the direction of all local edges, and the backward messages are traversed oppositely w.r.t. their direction when traversing the cycle according to its orientation. Bear in mind, however, that labelling the edges in a cycle as forward and backward is only of local significance. For example, in Fig. 1, the forward message $e$ in cycle $X$ is actually a backward one in cycle $Y$ (i.e., $e \in X^+$ and $e \in Y^-$).

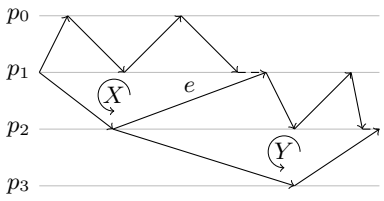


Fig. 1: An execution graph containing relevant cycles $X$, $Y$, and the combined cycle $X \oplus Y$, consisting of all edges except the oppositely oriented edge $e$.

**Definition 3 (ABC Synchrony Condition).** *Let $\Xi$ be a given rational number $\Xi > 1$, and let $G$ be the execution graph of an execution $\alpha$. Then, $\alpha$ is*

[2] The shadow graph $\bar{G}$ has the same set of vertices as $G$ and, for every edge in $G$, there is a corresponding undirectional edge in $\bar{G}$.

admissible in the ABC model *if, for every relevant cycle $Z$ in $G$, we have*

$$\frac{|Z^-|}{|Z^+|} < \Xi. \tag{2}$$

Note carefully that, compared to the purely asynchronous model, there is no other constraint in the ABC model: Only the ratio of the *number* of backward vs. forward messages in relevant cycles is constrained. There is no reference to end-to-end delays, no delay constraints whatsoever are put on individual messages, and none on messages in non-relevant cycles and isolated chains. Nevertheless, in Section 3, we will prove that the ABC synchrony condition is sufficient for simulating lock-step rounds, and hence for solving e.g. consensus by means of any synchronous consensus algorithm.[3]

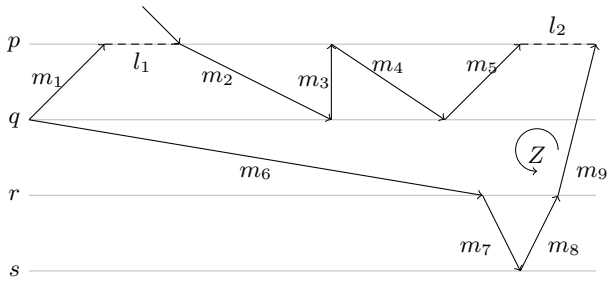


Fig. 2: A relevant cycle $Z$, where a causal chain $C_2 = m_1 l_1 m_2 \dots m_5 l_2$ is spanned by the "slow" message chain $C_1 = m_6 m_7 m_8 m_9$. Message $m_3$ has zero delay.

## 3 Clock Sychronization in the ABC Model

In this section, we show that the simple fault-tolerant tick generation algorithm introduced in [22] can be used for clock synchronization in the ABC model. It tolerates up to $f$ Byzantine process failures in a system of $n \geqslant 3f+1$ processes adhering to the ABC model. In Algorithm 1, every process $p$ maintains a local variable $k$ that constitutes $p$'s local clock as follows: Every process initially sets $k \leftarrow 0$ and broadcasts the message $(tick\ 0)$; for simplicity, we assume that a process sends messages also to itself. If a correct process $p$ receives $f+1$ $(tick\ \ell)$ messages (catch-up rule, line 3), it can be sure that at least one of them was sent by a correct process that has already reached clock value $l$. Therefore, $p$ can safely catch-up to $l$ and broadcast $(tick\ k+1),\dots,(tick\ l)$. If some process $p$ receives $n-f \geqslant 2f+1$ $(tick\ k)$ messages (advance rule, line 6) and thus advances its clock to $k+1$, it follows that at least $f+1$ of those messages will also be received by every other correct process, which then executes line 3. Hence, all

[3] Weaker ("eventual") variants of the ABC model are introduced in [19].

**Algorithm 1** Byzantine Clock Synchronization

```
1: VAR k: integer ← 0;
2: send (tick 0) to all [once];

   /* catch-up rule */
3: if received (tick l) from f + 1 distinct processes and l > k then
4:    send (tick k + 1),...,(tick l) to all [once];
5:    k ← l;
   /* advance rule */
6: if received (tick k) from n − f distinct processes then
7:    send (tick k + 1) to all [once];
8:    k ← k + 1;
```

correct processes will eventually receive $n - f$ (*tick* $k$) messages and advance their clocks to $k + 1$.

We will now prove that the algorithm guarantees progress of clocks and a certain synchrony condition, which can be stated in terms of consistent cuts in the execution graph. Note that using causality as a reference—rather than a common point in time, as in traditional clock synchronization—is natural in the time-free ABC model. Since the classic definition of consistent cuts does not take faulty processes into account, we will use the following correct-restricted version tailored to our execution graphs:

**Definition 4.** *Let $G$ be an execution graph and denote by $\stackrel{*}{\rightarrow}$ the reflexive and transitive closure of the edge relation $\rightarrow$. A subset $\mathcal{S}$ of events in $G$ is called* consistent cut*, if (1) for every correct process $p$, there is an event $\phi \in \mathcal{S}$ taking place at $p$, and (2) the set $\mathcal{S}$ is left-closed for $\stackrel{*}{\rightarrow}$; i.e., $\mathcal{S}$ contains the whole causal past of all events in $\mathcal{S}$.*

Given an event $\phi_p$ at process $p$, we denote by $C_p(\phi_p)$ the clock value *after* executing the computing step corresponding to $\phi_p$. Recall that the latter need not be correctly executed if $p$ is faulty. The clock value of a [correct] process $p$ in the frontier of a consistent cut $\mathcal{S}$ is denoted by $C_p(\mathcal{S})$; it is the last clock value of $p$ w.r.t. $\stackrel{*}{\rightarrow}$ in $\mathcal{S}$. Since it follows immediately from the code of Algorithm 1 that local clock values of correct processes are monotonically increasing, $C_p(\mathcal{S})$ is the maximum clock value at $p$ over all events $\phi_p \in \mathcal{S}$.

We first show that correct clocks make progress perpetually.

**Lemma 1 (One Step Progress).** *Let $\mathcal{S}$ be a consistent cut such that all correct processes $p$ satisfy $C_p(\mathcal{S}) \geqslant k$, for a fixed $k \geqslant 0$. Then there is a consistent cut $\mathcal{S}'$ where every correct process has set its clock to at least $k + 1$.*

*Proof.* If all correct processes $p_i$ have a (possibly distinct) clock value $k_i \geqslant k$ in the frontier of $\mathcal{S}$, the code of Algorithm 1 ensures that they have already sent (*tick* $k$). Since all messages in transit are eventually delivered, there must be a (not necessarily consistent) cut $\mathcal{S}''$, in the frontier of which every correct process

has received $n-f$ tick $k$ messages and thus set its clock to $k+1$. The left-closure of $\mathcal{S}''$ yields the sought consistent cut $\mathcal{S}'$. □

**Theorem 1 (Progress).** *In every admissible execution of Algorithm 1 in a system with $n \geqslant 3f+1$ processes, the clock of every correct process progresses without bound.*

*Proof.* The theorem follows from a trivial induction argument using Lemma 1, in conjunction with the fact that the cut $\mathcal{S}^0$ comprising the initial event $\phi_p^0$ of every process $p$ is trivially consistent and satisfies $C_p(\mathcal{S}^0) \geqslant 0$. □

**Lemma 2 (First Advance).** *If a correct process $q$ sets its clock to $k \geqslant 1$ in event $\psi_q$, then there is a correct process $p$ that sets its clock to $k$ using the advance rule in some event $\psi_p$ with $\psi_p \xrightarrow{*} \psi_q$.*

*Proof.* If $q$ uses the advance rule for setting its clock to $k$ in $\psi_q$, the lemma is trivially true. If $q$ uses the catch-up rule instead, it must have received $f+1$ (*tick* $k$) messages, at least one of which was sent by a correct process $q'$ in an event $\psi_{q'} \xrightarrow{*} \psi_q$. If $q'$ also sent its (*tick* $k$) via the catch-up rule (line 3), we apply the same reasoning to $q'$. Since every process sends (*tick* $k$) only once and there are only finitely many processes, we must eventually reach a correct process $p$ that sends (*tick* $k$) in event $\psi_p \xrightarrow{*} \psi_q$ via the advance rule. □

**Lemma 3 (Causal Chain Length).** *Assume that a correct process sets its clock to $k+m$, for some $k \geqslant 0$, $m \geqslant 0$, at some event $\phi'$, or has already done so. Then, there is a causal chain $D$ of length $|D| \geqslant m$ involving correct processes only, which ends at $\phi'$ and starts at some event $\phi$ where a correct process sets its clock to $k$ using the advance rule ($k \geqslant 1$) or the initialization rule ($k = 0$).*

*Proof.* Let $p$ be the process where $\phi'$ occurs. If $p$ has set its clock in some earlier computing step $\phi''' \xrightarrow{*} \phi'$, we just replace $\phi'$ by $\phi'''$ and continue with the case where $p$ sets its clock to $k+m$ in $\phi'$. If $p$ sets its clock in $\phi'$ using the catch-up rule, applying Lemma 2 yields a correct process that sets its clock to $k+m$ in an event $\psi \xrightarrow{*} \phi'$ using the advance rule. To prove Lemma 3, it hence suffices to assume that $p$ sets its clock to $k+m$ in $\phi'$ via the advance rule ($k+m \geqslant 1$) or the initialization rule ($k+m = 0$), as we can append the chains cut before to finally get the sought causal chain $D$.

The proof is by induction on $m$. For $m = 0$, the lemma is trivially true. For $m > 0$, at least $n-2f \geqslant f+1$ correct processes must have sent (*tick* $k+m-1$). Let $q$ be any such process, and $\phi''$ be the event in which (*tick* $k+m-1$) is sent. Since $q$ also sets its clock to $k+m-1$ at $\phi''$, we can invoke Lemma 2 in case $k+m-1 \geqslant 1$ to assure that the advance rule is used in $\phi''$; for $k+m-1 = 0$, the initialization rule is used in $\phi''$. We can hence apply the induction hypothesis and conclude that there is a causal chain $D'$ of length at least $m-1$ leading to $\phi''$. Hence, appending $q$'s (*tick* $k+m-1$) message [and the initially cut off chains] to $D'$ provides $D$ with $|D| \geqslant m$. □

The following Lemma 4 will be instrumental in our proof that Algorithm 1 maintains synchronized clocks. It reveals that when a correct process $p$ updates its clock value in some event $\phi'$, then all messages of correct processes of a certain lower tick value must have already been received by $p$, i.e., must originate from the causal cone of $\phi'$.

**Lemma 4 (Causal Cone).** *For some $k \geqslant 0$, suppose that $C_p(\phi') = k + 2\Xi$ at the event $\phi'$ of a correct process $p$. Then, for every $0 \leqslant \ell \leqslant k$, process $p$ has already received (tick $\ell$) from every correct process.*

*Proof.* The general proof idea is to show that the arrival of (*tick* $\ell$) in some event $\phi''$ after $\phi'$ would close a relevant cycle in which the synchrony assumption (2) is violated. See Fig. 3 for a graphical representation of the scenario described below.

Let $C_p(\phi') = k + 2\Xi$ and assume, for the sake of contradiction, that (*tick* $\ell$) from some correct process $q$ was not yet received by $p$ before or at $\phi'$, for some $\ell \leqslant k$. Consider the last message that $p$ received from $q$ before (or at) $\phi'$. If such a message exists, we denote its send event at $q$ as $\psi'$; otherwise, we simply define $\psi'$ to be the (externally triggered) initial computing step at $q$.

From Lemma 3, we know that there is a causal chain $D = \phi'_1 \to \cdots \to \phi'$ of length $|D| \geqslant k + 2\Xi - (\ell + 1)$, where a (*tick* $\ell + 1$) message is sent in $\phi'_1$ by some correct process $p_1$ via the advance rule and, by assumption, $C_p(\phi') = k + 2\Xi$. Since $\phi'_1$ executes the advance rule, $p_1$ must have received $n - f$ (*tick* $\ell$) messages to do so. Denoting by $0 \leqslant f' \leqslant f$ the actual number of faulty processes among the $n \geqslant 3f + 1$ ones, it follows that $n - f - f' \geqslant f + 1$ of these messages were sent by correct processes; we denote this set by $P_1$.

Since Theorem 1 ensures progress of all correct processes, there must be an event $\psi_1$, coninciding with or occuring after $\psi'$, in which $q$ broadcasts (*tick* $\ell$). Eventually, this message is received by $p$ in some event $\phi''$, which must be after $\phi'$ since by assumption (*tick* $\ell$) was not received before (or at) $\phi'$. Furthermore, we claim that $q$ receives at least $n - f' - f$ (*tick* $\ell$) messages from correct processes after (or at) event $\psi_1$; let $P_2$ be that set. Otherwise, $q$ would have received at least $n - f' - (n - f' - f) + 1 = f + 1$ (*tick* $\ell$) from correct processes by some event $\psi'_1 \stackrel{*}{\to} \psi_1$, and therefore would have broadcast (*tick* $\ell$) already in $\psi'_1$ according to the catch-up rule.

Since $P_1 \cup P_2$ is of size at most $2n - 2f' - 2f$ and we have only $n - f'$ correct processes, it follows by the pigeonhole principle that $2n - 2f' - 2f - (n - f') = n - 2f - f' \geqslant n - 3f > 0$ correct processes are in $P_1 \cap P_2$. Choose any process $p_0 \in P_1 \cap P_2$, which broadcasts its (*tick* $\ell$) in some event $\phi_0$. This message is received at $q$ in some event $\psi_2$, and at $p_1$ in event $\phi_1$.

It is immediately apparent from Fig. 3 that the causal chains $\phi_0 \to \phi_1 \stackrel{*}{\to} D \stackrel{*}{\to} \phi''$, $\phi_0 \to \psi_2$, $\psi_1 \stackrel{*}{\to} \psi_2$, and $\psi_1 \to \phi''$ form a relevant cycle $Z$: The number of backward messages is $|Z^-| = |D| + 1 \geqslant k - \ell + 2\Xi \geqslant 2\Xi$, since $\ell \leqslant k$; the number of forward messages $|Z^+|$ is 2. But this yields $\frac{|Z^-|}{|Z^+|} \geqslant \frac{2\Xi}{2} = \Xi$, contradicting the ABC synchrony assumption (2). □

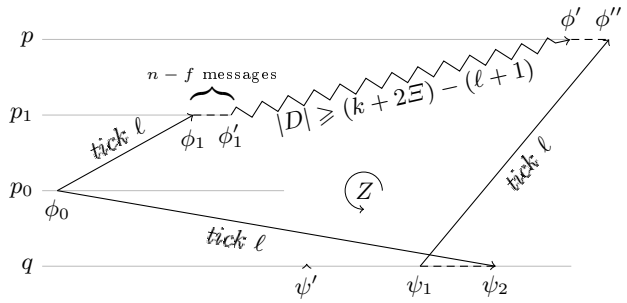


Fig. 3: Proof of Lemma 4

We can now easily prove that Algorithm 1 maintains the following synchrony condition:

**Theorem 2 (Synchrony).** *For any consistent cut $\mathcal{S}$ in an admissible execution of Algorithm 1 in a system with $n \geqslant 3f+1$ processes, we have $|C_p(\mathcal{S}) - C_q(\mathcal{S})| \leqslant 2\Xi$ for all correct processes $p$, $q$.*

*Proof.* Assume that the maximum clock value in the frontier of $\mathcal{S}$ is $k+2\Xi$, and let $p$ be a correct process with $C_p(\mathcal{S}) = k + 2\Xi$. From Lemma 4, we know that $p$ must have seen (*tick* $\ell$) from every correct process $q$ for any $\ell \leqslant k$. Since $\mathcal{S}$ is consistent, all the corresponding send events at $q$ must be within $\mathcal{S}$, such that $C_q(\mathcal{S}) \geqslant k$. □

Even though the ABC model is entirely time-free, we can immediately transfer the above synchrony property to real-time cuts according to [15], to derive the following theorem:

**Theorem 3 (Clock Precision).** *Let $C_p(t)$ denote the clock value of process $p$ at real-time $t$. For any time $t$ of an admissible execution of Algorithm 1 in a system with $n \geqslant 3f+1$ processes, we have $|C_p(t) - C_q(t)| \leqslant 2\Xi$ for all correct processes $p$, $q$.* □

Finally, we will show how to build a lock-step round simulation in the ABC model atop of Algorithm 1. A lock-step round execution proceeds in a sequence of rounds $r = 1, 2, \ldots$, where all correct processes take their round $r$ computing steps (consisting of receiving the round $r-1$ messages[4], executing a state transition, and broadcasting the round $r$ messages for the next round) exactly at the same time.

We use the same simulation as in [22], which just considers clocks as phase counters and introduces rounds consisting of $2\Xi$ phases. Algorithm 2 shows the code that must be merged with Algorithm 1; the round $r$ messages are piggybacked on (*tick* $k$) messages every $2\Xi$ phases, namely, when $k/(2\Xi) = r$.

[4] For notational convenience, we enumerate the messages with the index of the previous round.

**Algorithm 2** Lock-Step Round Simulation

```
1:  VAR r: integer ← 0;
2:  call start(0);

3:  Whenever k is updated do
4:  if k/(2Ξ) = r + 1 then
5:     r ← r + 1
6:     call start(r)

7:  procedure start(r:integer)
8:     if r > 0 then
9:        read round r − 1 messages
10:       execute round r computation
11:    send round r messages
```

The round $r$ computing step is encapsulated in the function start($r$) in line 7; start(0) just sends the round 0 messages that will be processed in the round 1 computing step.

To prove that this algorithm achieves lock-step rounds, we need to show that all round $r$ messages from correct processes have arrived at every correct process $p$ before $p$ enters round $r+1$, i.e., executes start($r+1$).

**Theorem 4 (Lock-Step Rounds).** *In a system with $n \geqslant 3f+1$ processes, Algorithm 2 merged with Algorithm 1 correctly simulates lock-step rounds in the ABC model.*

*Proof.* Suppose that a correct process $p$ starts round $r+1$ in event $\phi$. By the code, $C_p(\phi) = k$ with $k/(2\Xi) = r+1$, i.e., $k = 2\Xi r + 2\Xi$. By way of contradiction, assume that the round $r$ message, sent by some correct process $q$ in the event $\psi$, arrives at $p$ only after $\phi$. By the code, $C_q(\psi) = k'$ with $k'/(2\Xi) = r$, i.e., $k' = 2\Xi r$. However, Lemma 4 reveals that $p$ should have already seen (*tick* $2\Xi r$) from $q$ before event $\phi$, a contradiction. □

**Remark:** We note that the above proof(s) actually establish *uniform* [10] lock-step rounds, i.e., lock-step rounds that are also obeyed by faulty processes until they fail for the first time: If the messages sent by faulty processes also obey the ABC synchrony condition (2), then the proof of the key Lemma 4 actually establishes a uniform causal cone property: Assuming that (i) process $q$ performs correctly up to and including at least one more step after event $\psi'$, and (ii) $p$ works correctly up to and including event $\phi'$, then $p$ would receive (though not necessarily process) the message from $q$ in $\phi''$, thereby closing a relevant cycle that violates $\Xi$. Hence, $p$ must have received all messages from its causal cone by $\phi'$ already, which carries over to a uniform version of Theorem 4.

## 4 Model Indistinguishability

In this section, we will develop a non-trivial "model indistinguishability" argument in order to show that any algorithm designed for the $\Theta$-Model [4, 21, 22] also works correctly in the ABC model. It is non-trivial, since there are (many) admissible ABC executions which are *not* admissible in the $\Theta$-Model. Nevertheless, no simulation will be involved in our argument; the original algorithms can just be used "as is" in the ABC model. This "timing invariance" of algorithms and their properties confirms again that timing constraints are not really essential for solving certain distributed computing problems.

More specifically, provided that $\Xi < \Theta$, we will show that every algorithm designed and proved correct for the $\Theta$-Model preserves all its timing-independent properties when executed in the ABC model. Note that the algorithms analyzed in Section 3 belong to this class.[5]

Like the ABC model, the $\Theta$-Model [4, 21, 22] is a message-driven model, without real-time clocks, and hence also relies on end-to-end delays. If $\tau^+(t)$ resp. $\tau^-(t)$ denotes the (unknown) maximum resp. minimum delay of all messages in transit system-wide between correct processes at time $t$, it just assumes that there is some $\Theta > 1$ such that

$$\frac{\tau^+(t)}{\tau^-(t)} \leqslant \Theta \tag{3}$$

at all times $t$ in all admissible executions. In the simple static $\Theta$-Model (which is sufficient for our model indistinguishability argument, since it has been shown in [22] to be equivalent to the general $\Theta$-Model from the point of view of algorithms), it is assumed that there are (unknown) upper resp. lower bounds $\infty > \tau^+ \geqslant \tau^+(t)$ resp. $0 < \tau^- \leqslant \tau^-(t)$ on the end-to-end delays of all correct messages in all executions, the ratio of which matches the (known) $\Theta = \frac{\tau^+}{\tau^-}$.

Formally, fix some algorithm $A$ and let $ASYNC$ be the set of executions of $A$ running in a purely asynchronous message-driven system; note that we consider *timed executions* here, i.e., executions along with the occurence times of their events. A *property* $P$ is a subset of the admissible executions in $ASYNC$, i.e., a property is defined via the executions of $A$ that satisfy it. Let $\mathcal{M}$ be the set of admissible executions of $A$ in some model $M$ that augments the asynchronous model, by adding additional constraints like the ABC synchrony condition. Clearly, $\mathcal{M}$ is the intersection of some model-specific safety and liveness properties in $ASYNC$. We say that an execution $\alpha$ is in model $M$ if $\alpha \in \mathcal{M}$, i.e., if $\alpha$ is admissible in $M$. If $\mathcal{M} \subseteq P$, we say that $A$ *satisfies* property $P$ in the model $\mathcal{M}$. A property $P$ is called *timing-independent*, if $\alpha \in P \Rightarrow \alpha' \in P$ for every pair of causally equivalent executions $\alpha$, $\alpha'$, i.e., executions where $G_\alpha = G_{\alpha'}$.

[5] It is not possible to derive our results from the $\Theta$-based analysis in [22], however, since this would require carrying over timing-dependent properties. And indeed, the $\Theta$-variant [22] of our synchronizer (Algorithm 2) requires rounds consisting of $3\Theta$ phases, rather than $2\Xi$ phases as in the ABC model.

First, using a trivial model-indistinguishability argument, it is easy to show that properties of an algorithm proved to hold in the ABC model $M_{ABC}$ also hold in the $\Theta$-Model $M_\Theta$, for any $\Theta < \Xi$: The following Theorem 5 exploits the fact that the relevant cycles in the execution graph $G_\alpha$, corresponding to an admissible execution $\alpha$ in the $\Theta$-Model, also satisfy the ABC synchrony condition (2), i.e., that $\alpha$ is an admissible execution in the ABC model as well.

**Theorem 5.** *For any $\Theta < \Xi$, it holds that $\mathcal{M}_\Theta \subseteq \mathcal{M}_{ABC}$. Hence, if an algorithm satisfies a property $P$ in the ABC model, it also satisfies $P$ in the $\Theta$-Model.*

*Proof.* If $Z$ is any relevant cycle in $G_\alpha$, then no more than $|Z^+|\Theta$ backward messages can be in $Z$; otherwise, at least one forward-backward message pair would violate (3). It follows that $|Z^-|/|Z^+| \leqslant \Theta < \Xi$ as required. Hence, $\mathcal{M}_\Theta \subseteq \mathcal{M}_{ABC} \subseteq P$, since the algorithm satisfies $P$ in the ABC model. □

The converse of Theorem 5 is not true, however: The time-free synchrony assumption (2) of the ABC model allows arbitrary small end-to-end delays for individual messages, violating (3) for every $\Theta$. From a timing perspective, the ABC model is indeed strictly weaker than the $\Theta$-Model, hence $\mathcal{M}_{ABC} \not\subseteq \mathcal{M}_\Theta$. Nevertheless, Theorem 6 below shows that, given an arbitrary finite execution graph $G$ in $\mathcal{M}_{ABC}$, it is always possible to assign end-to-end delays $\in (1, \Xi)$ to the individual messages *without changing the event order* at any process. Let $\tau$ be such a delay assignment function, and $G^\tau$ be the weighted execution graph obtained from $G$ by adding the assigned delays to the messages. Since $\Theta$-algorithms are message-driven, without real-time clocks, $G$ and $G^\tau$ are indistinguishable for every process. Consequently, an algorithm that provides certain timing-independent properties when being run in the $\Theta$-Model also maintains these properties in the ABC model, see Theorem 8.

**Theorem 6.** *For every finite ABC execution graph $G$, there is an end-to-end delay assignment function $\tau$, such that the weighted execution graph $G^\tau$ is causally equivalent to $G$ and all messages in $G^\tau$ satisfy* (3).

*Proof.* The (quite involved) proof, which utilizes a non-standard cycle-space of a graph and an algebraic treatment of a system of linear inequalities using a variant of Farkas' lemma, can be found in [19]. □

In order to formally prove the claimed "model indistinguishability" of the ABC model and the $\Theta$-Model, we proceed with the following Lemma 5. It says that processes cannot notice any difference in finite prefixes in the ABC model and in the $\Theta$-Model, and therefore make the same state transitions.

**Lemma 5 (Safety Equivalence).** *If an algorithm satisfies a timing-independent safety property $S$ in the $\Theta$-Model, then $S$ also holds in the ABC model, for any $\Xi < \Theta$.*

*Proof.* Suppose, by way of contradiction, that there is a finite prefix $\beta$ of an ABC model execution $\alpha \in \mathcal{M}_{ABC}$, where $S$ does not hold. Furthermore, let $\beta'$

be a finite extension of $\beta$ such that all messages sent by correct processes in $\beta$ arrive in $\beta'$, and denote the execution graph of $\beta'$ by $G_{\beta'}$. From Theorem 6, we know that there is a delay assignment $\tau$ such that the synchrony assumption (3) of the $\Theta$-Model is satisfied for all messages in the timed execution graph $G^\tau_{\beta'}$, while the causality relation in $G_{\beta'}$ and $G^\tau_{\beta'}$ (and since $G^\tau_{\beta'} \supseteq G^\tau_\beta$ also in $G_\beta$ and $G^\tau_\beta$) is the same.

We will now construct an admissible execution $\gamma$ in the $\Theta$-Model, which has the same prefix $G^\tau_{\beta'}$: If $t$ is the greatest occurrence time of all events in $G^\tau_{\beta'}$, we simply assign an end-to-end delay of $\tau^+$ to all messages still in transit at time $t$ and to all messages sent at a later point in time. Note that $\gamma$ may be totally different from the ABC-execution $\alpha$ w.r.t. the event ordering *after* the common prefix $\beta'$. Anyway, $\gamma$ is admissible in the $\Theta$-Model since (3) holds for all messages, but violates $S$, which provides the required contradiction. □

Unfortunately, we cannot use the same reasoning for "transfering" liveness properties, since finite prefixes of an execution are not sufficient to show that "something good" eventually happens. Nevertheless, Theorem 7 below reveals that all properties satisfiable by an algorithm in the $\Theta$-Model are actually safety properties, in the following sense: For every property $P$ (which could be a liveness property like termination) satisfied by $A$ in $M_\Theta$, there is a (typically stronger) safety property $P' \subseteq P$ (like termination within time $X$) that is also satisfied by $A$ in $M_\Theta$. Hence, there is no need to deal with liveness properties here at all.

For our proof, we utilize the convenient topological framework introduced in [2], where safety properties correspond to the closed sets of executions in *ASYNC*, and liveness properties correspond to dense sets. If a model $M$ is determined solely by safety properties $S_1, \ldots, S_k$, then the set $\mathcal{M} = \bigcap_{i=1}^k \mathcal{S}_i$—and therefore the model $M$—is *closed*.

**Theorem 7 (Safety-Only in Closed Models).** *Let $M$ be a closed model augmenting the asynchronous model, and let $\mathcal{M} \subseteq ASYNC$ be the set of all admissible executions of an algorithm $A$ in $M$. To show that $A$ satisfies some arbitrary property $P$ in $M$, it suffices to show that $A$ satisfies the property $P' = P \cap \mathcal{M}$, which is a safety propery.*

*Proof.* Suppose that $A$ satisfies some property $P \subseteq ASYNC$ in $M$, i.e., $\mathcal{M} \subseteq P$. Then, $\mathcal{M} = \mathcal{M} \cap P$ and since $\mathcal{M}$ is closed, it follows that $P' = \mathcal{M} \cap P$ is closed (in *ASYNC*) as well. But this is exactly the definition of a safety property $P' \subseteq ASYNC$ and, since $\mathcal{M} \subseteq P' \subseteq P$, it indeed suffices to show that A satisfies $P'$ in $M$. □

**Lemma 6.** *The ABC model and the $\Theta$-Model are both closed.*

*Proof.* We just need to show that the set $\mathcal{M}_\Theta$ resp. $\mathcal{M}_{ABC}$ of executions in the $\Theta$-Model resp. in the ABC model is closed. If some execution violated the end-to-end timing assumption (3) of the $\Theta$-Model resp. the ABC synchrony condition (2), there would be a finite prefix within which this violation has happened. This characterizes a safety property and hence a closed set in *ASYNC*. □

Theorem 7 in conjunction with Lemma 6 reveals that every property satisfiable in the $\Theta$-Model is a safety property. Hence, Lemma 5 finally implies Theorem 8, which complements Theorem 5.

**Theorem 8.** *All timing-independent properties satisfied by an algorithm in the $\Theta$-Model also hold in the ABC model, for any $\Xi < \Theta$.* □

## 5 Relation to the Classic Partially Synchronous Model

In this section, we relate the ABC model to the perpetual partially synchronous model (ParSync) introduced in [7]. ParSync stipulates a bound $\Phi$ on relative computing speeds and a bound $\Delta$ on message delays, relative to an (external) discrete global clock, which ticks whenever a process takes a step: During $\Phi$ ticks of the global clock, every process takes at least one step, and if a message $m$ was sent at time $k$ to a process $p$ that subsequently performs a receive step at or after time $k + \Delta$, $p$ is guaranteed to receive $m$.

First of all, we note that the ABC model and ParSync are equivalent in terms of solvability of timing-independent problems in fully-connected networks: In [22], it was shown that the $\Theta$-Model and ParSync are equivalent is this regard: Since the synchrony parameters $\Phi$, $\Delta$ of the ParSync model imply bounded (and non-zero) end-to-end delays, any $\Theta$-algorithm can be run in a ParSync system if $\Theta = \Theta(\Phi, \Delta)$ is chosen sufficiently large. Conversely, using the lock-step round simulation for the $\Theta$-Model provides a "perfect" ParSync system ($\Phi = 1$ and $\Delta = 0$), which obviously allows to execute any ParSync algorithm atop of it. The claimed equivalence thus follows from the model indistinguishability of the ABC model and the $\Theta$-Model established in Section 4.

This problem equivalence does not imply that the models are indeed equivalent, however. First, as shown below, there are problems that can be solved in ABC model but not in ParSync in case of not fully-connected networks. Moreover, whereas we can choose $\Xi$ such that every execution of a message-driven algorithm in ParSync with $\Phi$, $\Delta$ is also admissible in the ABC model for some $\Xi > \Theta(\Phi, \Delta)$, we can even conclude from $\mathcal{M}_{\text{ABC}} \supset \mathcal{M}_{\Theta}$ that some ABC executions cannot be modeled in ParSync. To investigate this issue in more detail, we use the taxonomy of partially synchronous models introduced in [6], which delimits the exact border between consensus solvability and impossibility: It distinguishes whether ($c$) communication is synchronous ($\Delta$ holds) or asynchronous, whether ($p$) processes are synchronous ($\Phi$ holds) or asynchronous, whether ($s$) steps are atomic (send+receive in a single step) or non-atomic (separate send and receive steps), whether ($b$) send steps can broadcast or only unicast, and whether ($m$) message delivery is (globally) FIFO ordered or out-of-order.

We will argue below that the ABC model model must be mapped to the case of asynchronous communication, asynchronous processes, atomic steps, broadcast send and out-of-oder delivery. Using the corresponding "binary encoding" ($c = 0, p = 0, s = 1, b = 1, m = 0$) in [6, Table 1], it turns out that consensus is not solvable in the resulting model. Of course, the apparent contradiction to

the solvability of consensus in the ABC model is due to the ABC synchrony condition, which cannot be properly expressed in the framework of [6].

**Asynchronous communication and asynchronous processes:** Consider a 2-player game where the Prover first chooses $\Xi$ and the Adversary, knowing $\Xi$, chooses a pair $(\Phi, \Delta)$. Finally, the Prover has to choose an execution satisfying (2) for $\Xi$; the Prover wins iff this execution violates the adversary-chosen parameters $(\Phi, \Delta)$. The Prover has a winning strategy: It suffices to choose any execution containing a relevant cycle as shown in Fig. 2, which respects (2) but lets $|Z^-|$ be greater than both $\Phi$ and $\Delta$: While the (slow) message $m_6$ from $q$ to $r$ is in transit, process $q$ executes more than $\Delta$ steps. Moreover, neither process $r$ nor $s$ execute a step during the more than $\Phi$ steps of $q$. As a consequence, both communication and processes must be considered asynchronous ($c = 0, p = 0$).

**Atomic steps and broadcast:** Whereas it is clear that out-of-order delivery ($m = 0$) makes it more difficult to solve problems, one may be wondering whether the "favorable" choices $s = 1$ and $b = 1$, rather than the ABC synchrony condition, make consensus solvable in the ABC model. [6, Table 1] reveals that this is not the case: All entries corresponding to $p = 0, c = 0, m = 0$ are the same (consensus impossible), irrespectively of the choice of $b$ and $s$. And indeed, the assumption of atomic receive+broadcast steps in the ABC model's definition in Section 2 is just a simplifying abstraction: Every non-atomic broadcast execution (= multiple unicast steps) can be mapped to a causally equivalent atomic receive+broadcast step execution with appropriately adjusted end-to-end delays.[6]

Another major difference between ParSync and the ABC model results from the cumulative and non-global character of the ABC synchrony condition. Since (2) needs to hold only in relevant cycles, which are in fact defined by the specific algorithm employed, the ABC model is particularly suitable for modeling systems with not fully-connected communication graphs: For choosing $\Xi$, only the cumulative end-to-end delay ratio over certain paths counts.

Consider the execution shown in Fig. 5, for example, which corresponds to a system where process $q$ exchanges messages directly with $p$ (over a 1-hop path $P_{qpq}$), and indirectly with $s$ (over a 2-hop path $P_{qrsrq}$ via $r$). As long as the sum of the delays along $P_{qrsrq}$ is less than the cumulative delay of $\Xi$ instances of $P_{qpq}$, the individual delays along the links between $q, r$ and $r, s$ are totally irrelevant. In the VLSI context, for instance, this gives more flexibility for place-and-route, as well as some robustness against dynamic delay variations. By contrast, in ParSync, very conservative values of $\Phi$, $\Delta$ would be needed to achieve a comparable flexibility; obviously, this would considerably degrade the achievable performance system-wide.

In case of not fully-connected networks, there are even situations which cannot be modeled in ParSync at all. Consider the message-pattern given in Fig. 4 in a system with $\Xi = 4$, for example: The ABC synchrony condition ensures FIFO order of the messages sent from $p_2$ to $q_1$, even when their delay is unbounded (and may even continuously grow, as e.g. in a formation of fixed-constellation

[6] The ABC model can hence also be used for making classic distributed algorithms results applicable to non-atomic models like the Real-Time Model introduced in [16].

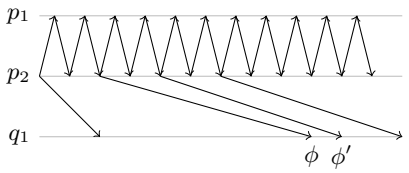


Fig. 4: An execution of a system implementing bounded-size FIFO channels. If the order of $\phi$ and $\phi'$ changed, there would be a relevant cycle violating (2) for $\Xi = 4$.

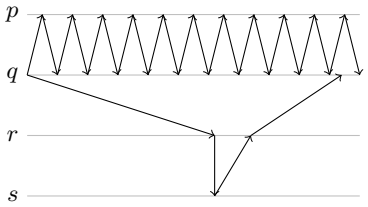


Fig. 5: The long delay on the link between $q$ and $r$ is compensated by the fast delay on the link between $r$ and $s$.

clusters of spacecraft that move away from each other): If there was a reordering of $\phi$ and $\phi'$, a relevant cycle with $\Xi = 5$ would be formed, which is impossible. Note that processes $p_1$, $p_2$ make unbounded progress while a message to $q_1$ is in transit here. Hence, as in the example of Fig. 2 mentioned before, the problem cannot be solved in ParSync. Clearly, such message ordering capabilities are very useful in practice, e.g., for implementing stable identifiers, bounded-message size, single source FIFOs etc.

## 6 Conclusions and Future Work

We have introduced a novel partially synchronous system model, the ABC model, which is completely time-free and thus rests on a causality-based synchrony condition only. We showed that it is sufficiently strong for implementing lock-step rounds and, hence, for solving any important distributed computing problem, including consensus. We also proved that algorithms designed for the $\Theta$-Model also work correctly in the ABC model. Part of our future work is devoted to exploiting the ABC model in the chase for the weakest system model for solving consensus, and to the analysis of the ABC model's coverage in real systems, in particular, VLSI Systems-on-Chip.

### Acknowledgments

We are indebted to Martin Biely, Josef Widder, Martin Hutle, Matthias Függer, Heinrich Moser, and Bernadette Charron-Bost for their contributions to the ABC model.

## References

1. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, *Communication-efficient leader election and consensus with limited link synchrony*, Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC'04) (St. John's, Newfoundland, Canada), ACM Press, 2004, pp. 328–337.

2. B. Alpern and Fred B. Schneider, *Defining liveness*, Tech. report, Ithaca, NY, USA, 1984.
3. Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, *Bounds on the time to reach agreement in the presence of timing uncertainty*, Journal of the ACM (JACM) **41** (1994), no. 1, 122–152.
4. M. Biely and J. Widder, *Optimal message-driven implementation of omega with mute processes*, Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006) (Dallas, TX, USA), LNCS, vol. 4280, Springer Verlag, November 2006, pp. 110–121.
5. Tushar Deepak Chandra and Sam Toueg, *Unreliable failure detectors for reliable distributed systems*, Journal of the ACM **43** (1996), no. 2, 225–267.
6. D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, Journal of the ACM **34** (1987), no. 1, 77–97.
7. C. Dwork, N. Lynch, and L. Stockmeyer, *Consensus in the presence of partial synchrony*, Journal of the ACM **35** (1988), no. 2, 288–323.
8. Christof Fetzer, Ulrich Schmid, and Martin Süßkraut, *On the possibility of consensus in asynchronous systems with finite average response times*, Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05) (Washington, DC, USA), IEEE Computer Society, June 2005, pp. 271–280.
9. Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM **32** (1985), no. 2, 374–382.
10. V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, Distributed Systems (Sape Mullender, ed.), Addison-Wesley, 2nd ed., 1993, pp. 97–145.
11. Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou, *Brief announcement: Chasing the weakest system model for implementing omega and consensus*, Proceedings Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (formerly Symposium on Self-stabilizing Systems) (SSS 2006) (Dallas, USA), LNCS, Springer Verlag, Nov. 2006, pp. 576–577.
12. Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM **21** (1978), no. 7, 558–565.
13. D. Malkhi, F. Oprea, and L. Zhou, *$\Omega$ meets paxos: Leader election and stability without eventual timely links*, Proceedings of the 19th Symposium on Distributed Computing (DISC'05), LNCS, vol. 3724, Springer Verlag, 2005, pp. 199–213.
14. F. Mattern, *Virtual time and global states of distributed systems*, Proceedings of the International Workshop on Parallel and Distributed Algorithms, Elsevier Science Publishers B.V, 1989, pp. 215–226.
15. F. Mattern, *On the relativistic structure of logical time in distributed systems*, in Parallel and Distributed Algorithms, Elsevier Science Publishers B.V, 1992, pp. 215–226.
16. H. Moser and U. Schmid, *Optimal clock synchronization revisited: Upper and lower bounds in real-time systems*, Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'06) (Bordeaux & Saint-Emilion, France), LNCS 4305, Springer Verlag, Dec 2006, pp. 95–109.
17. A. Mostefaoui, E. Mourgaya, M. Raynal, and C. Travers, *A time-free assumption to implement eventual leadership.*, Parallel Processing Letters, 16 (2006), pp. 189–208.
18. Stephen Ponzio and Ray Strong, *Semisynchrony and real time*, Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92) (Haifa, Israel), November 1992, pp. 120–135.
19. Peter Robinson and Ulrich Schmid, *The Asynchronous Bounded-Cycle Model*, Research Report 24/2008, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2008.

20. Paul M.B. Vitányi, *Distributed elections in an archimedean ring of processors*, Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM Press, 1984, pp. 542–547.
21. J. Widder, G. Le Lann, and U. Schmid, *Failure detection with booting in partially synchronous systems*, Proceedings of the 5th European Dependable Computing Conference (EDCC-5), LNCS, vol. 3463, Springer Verlag, April 2005, pp. 20–37.
22. J. Widder and U. Schmid, *Achieving synchrony without clocks*, Research Report 49/2005, Technische Universität Wien, Institut f. Technische Informatik, (submitted).