A DISTRIBUTED ALGORITHM FOR UNSTABLE GLOBAL PREDICATE EVALUATION WITH APPROXIMATELY SYNCHRONIZED CLOCKS

SHILPA BANSOD AND JEAN MAYO

Abstract. The ability to evaluate predicates over the global state of a computation is fundamental to distributed application development and control. Algorithms structured on the use of approximately synchronized clocks, for which there is a known bound on the difference of the readings of any two clocks at the same instant of time, are able to detect the occurrence of certain predicates at a point in time during a computation, when this is difficult or impossible otherwise.

Existing time-based algorithms for the detection of unstable predicates, for which the truth can vary arbitrarily over time, are centralized. Additionally, these algorithms are able to detect predicates only when they have again become false, or they require a known bound on message delivery delay. We propose an algorithm designed to be more scalable. The algorithm is distributed. Each process only retains a history of its own state that is related to truth of the global predicate. Further, local state information that cannot contribute to the truth of the global predicate is periodically removed. Message delay is assumed to be bounded, but the bound need not be known.

When unstable predicates are detected using approximately synchronized clocks, there is some minimum interval for which the predicate must be globally true to ensure its detection using clock values. This arises from the potential difference in process clock values at a single point in time. In this paper, we establish a lower bound on the interval a predicate must be globally true to ensure its detection using approximately synchronized clocks.

1. Introduction

The ability to evaluate predicates over the global state of a computation is fundamental to the development and control of distributed applications. Distributed debugging is facilitated by the ability to set a breakpoint at a particular point within a computation. Application control may require the ability to detect certain conditions, such as entry of all processes into a barrier, so that appropriate action can
be taken. Monitoring is predicated on the ability to detect and record global states of interest, such as entry into a critical section by one or more system processes.

Most work on predicate evaluation centers around evaluation of predicates over *consistent global states* within the distributed computation. A global state is a collection of process and channel states, one from each process and channel within the system. Informally, a global state is consistent if the collection of states *could* have occurred at the same instant of time within the computation. Determination of whether a particular set of states could have occurred during the same instant of time is based upon the use of Lamport’s happened-before relation [1]. This relation orders local process events globally without the use of a common time-base. Global ordering of local events arises from the message passing that occurs among processes. As transmission of a message takes non-zero time, it is known that any state occurring in the sender prior to the send of a particular message, must have happened before any state in the receiver occurring after the receipt of the same message. In the absence of message passing, any collection of process states is consistent, although a particular execution of the system cannot pass through all of these global states.

A consistent global state has the property that even if it did not occur at an instant in time during some observed execution, then it could occur in a subsequent execution. This is beneficial in evaluation of global state predicates which should never be true within the computation, such as violation of mutual exclusion. However, it is sometimes important to detect whether or not certain global predicates *actually* occurred at some point in time. When the evaluation is over consistent global states, it is sometimes impossible to detect the actual occurrence of the predicate, since it is not known whether or not a computation actually passed through a particular consistent global state. This is especially problematic in distributed systems that interact with the physical environment. In these systems, changes in the truth of global predicates are commonly rooted in time, and not in the message passing that occurs among processes within the system.
We present an algorithm that uses a common time base to order local events globally. Although no common time base exists naturally within a distributed system, one can be created by synchronizing the processor clocks so that the difference in the readings of any two clocks at a single instant of time (the clock skew) is kept within some known bound. This is achieved in hardware, software, or hybrid combinations of hardware and software [2]. Hardware based on use of the Global Positioning System (GPS) allows the clocks of geographically-dispersed processors to be tightly synchronized. Through a combination of hardware and software, the clocks of processors world-wide can be synchronized within a few milliseconds or less [3]. By taking advantage of a common time base, the actual occurrence of certain predicates can be detected, when this would not be possible otherwise [4].

We focus on the detection of unstable predicates, for which the truth can vary arbitrarily over time. Existing time-based algorithms are centralized; a single process collects and evaluates information from each system process on the truth of the global predicate [5, 6, 4]. These algorithms are able to detect predicates only when they have again become false, or they require a known bound on message delivery delay. The algorithm presented in this paper is designed to be more scalable. The algorithm is distributed; each process only retains a history of its own state that is related to truth of the global predicate and any process can detect the predicate. Further, local state information that cannot contribute to the truth of the global predicate is periodically removed. Message delay is assumed to be bounded, but the bound need not be known.

It is impossible to implement perfectly synchronized clocks[7]. There is then some minimum duration, proportional to the clock skew, over which a predicate must be globally true to ensure its detection using approximately synchronized clocks. In this paper, we establish this lower bound.

2. Related Work

A number of modalities have been defined for the evaluation of predicates over the global state of a distributed computation. Stoller presents a thorough discussion
of these modalities [4]. We focus here only on work related to detecting, using approximately synchronized clocks, the actual truth of a global state predicate at an instant of time during a computation.

Marzullo and Neiger presented algorithms for evaluation of predicates using approximately synchronized clocks [5]. Their algorithms are centralized. Processes send global states, and corresponding vector clock [8, 9] and approximately synchronized clock readings, to a central monitor process. The central monitor process collects these states and reconstructs all possible executions (via a lattice in which each point is a global state that could have occurred at an instant of time). If a predicate is globally true for a sufficient duration, then all executions will contain a global state in which the predicate is true. The algorithm supports detection of arbitrary predicates.

Mayo and Kearns presented an algorithm for the runtime detection of the actual occurrence of a global predicate [6]. Their algorithm is centralized. A queue is maintained for each process. The queue contains local clock values for the point at which a predicate becomes true, and when it again becomes false. Message delay is assumed to be bounded. Hence, the truth of a predicate can be detected without requiring receipt by the monitor of messages indicating the predicate has again become false (or periodic messages indicating the predicate is still true). The algorithm supports evaluation of conjunctive predicates over the local process states.

Stoller defined the modality $\text{Inst}(\phi)$ for evaluation of a global state predicate $\phi$. When $\text{Inst}(\phi)$ holds, then $\phi$ held over a global state that occurred at an instant of time during the computation [4]. He also presents an algorithm for detection of $\text{Inst}(\phi)$. The presented algorithm is centralized and handles the case of varying clock skew. Evaluation is performed over queues that contain process states and corresponding local clock values. The algorithm supports evaluation of arbitrary predicates.

The contribution of this paper is two-fold. First, an algorithm that is more scalable than existing time-based algorithms for the detection of the truth of a
conjunctive global predicate at some point in time during the computation is presented. The algorithm is distributed; a token circulates through the system processes to detect the truth of the global predicate. Each process only maintains a local time-history of the truth of the local component of a conjunctive global predicate. The token circulates continuously, so that local state information that cannot contribute to the truth of the global predicate is periodically removed. Secondly, a lower bound on the duration for which a conjunctive predicate must be globally true in order to be detected using approximately synchronized clocks is established.

3. A Distributed Algorithm for Global Predicate Evaluation

3.1. System Model

We consider a distributed system comprised of a set of reliable processes \( \{P_1, P_2, \ldots, P_n\} \), which communicate solely via message passing. As a notational convenience, we let \( SYS \) denote the set of process indices \( \{1, 2, \ldots, n\} \). Each process identifier is assumed to be unique within the system.

Each process in the system passes through a sequence of local states. The state of a process changes in response to events, which can be either a local action, or the sending or receipt of a message. A collection of local states, one from each process, comprises a global state.

The evaluation is token-based. A single token circulates through a logical cycle constructed on the set of processes. \( P_{(i+1) \mod n} \) is the successor of \( P_i \) in this cycle. No restriction is made on the routing of messages unrelated to the detection. We assume that this token is first generated by a pre-determined, but arbitrary, process within the system. The communication channels are assumed to be reliable, that is, no messages are lost, corrupted, or duplicated. Messages are subject to a bounded, but arbitrary and unknown, delivery delay.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{clock}_i )</td>
<td>( P_i )'s current clock reading</td>
</tr>
<tr>
<td>( \text{next}_i )</td>
<td>( P_i )'s successor, ((i + 1) \mod n), in the virtual ring</td>
</tr>
<tr>
<td>( \text{state}_i )</td>
<td>( P_i )'s state; initially \textit{waiting}</td>
</tr>
<tr>
<td>( T )</td>
<td>End time-stamp of interval at head of ( \gamma_i ) if interval is closed, else value of ( \text{clock}_i ); initially -1</td>
</tr>
<tr>
<td>( \gamma_i )</td>
<td>Queue of intervals over which ( A_i ) was true; each interval has the form ( \langle T_S, T_F \rangle ); queue is sorted in increasing order by time-stamp ( T_S ); initially empty</td>
</tr>
</tbody>
</table>

**Insert(\( \gamma_i, e \))** Inserts element \( e \) at the end of queue \( \gamma_i \)

**HeadOf(\( \gamma_i \))** Returns the first element from \( \gamma_i \); if \( q \) is empty, returns \( \langle -1, -1 \rangle \)

**CloseInterval(\( \gamma_i \))** Closes open interval at end of \( \gamma_i \) by setting the second interval component value to \( \text{clock}_i \)

**STime(\( I_i \))** Returns the component \( T_S \) of \( I_i \), where \( I_i \in \gamma_i \)

**ETime(\( I_i \))** Returns -1 if the component \( T_F \) of \( I_i \) is zero; else returns \( T_F \), where, \( I_i \in \gamma_i \)

**RecvToken()** Routine to process received token; defined in Figure 3.2

**Token(\( p, T_{S_{max}}, T_{F_{min}}, T_{S_{clean}} \))** Detection token; initially, \( p, T_{S_{max}}, T_{F_{min}}, \) and \( T_{S_{clean}} \) have value -1

Figure 3.1: Algorithm State and Supported Functions for Process \( P_i \)

SIU 2004
Processes share no common memory or common clock. Each process $P_i$ is assumed to have access to a local clock represented by the non-decreasing real-valued function $C_i$, where $C_i(t)$ represents the time on the clock of $P_i$ at real time instant $t$. In this paper, we adopt the notational convention that times denoted with uppercase letters are process clock values; times denoted by lowercase letters are real times. The inverse mapping $C_i^{-1}(T)$ represents the set of real time instants at which $C_i$ reads the value $T$. $\text{Inf}(C_i^{-1}(T))$ and $\text{sup}(C_i^{-1}(T))$ return the earliest and latest real time instants, respectively, when $C_i$ reads the value $T$. We make the assumption that all process clocks are approximately synchronized within some known bounded error $\epsilon$. We also assume that a local clock has sufficient resolution to distinguish between the times of any two events related to the algorithm at the site of the clock. These properties are formally stated by the following Clock Axioms:

CA1 For all $P_i$, $i \in \text{SYS}$, if $t \geq t'$ then $C_i(t) \geq C_i(t')$ and if $C_i(t) > C_i(t')$ then $t > t'$.

CA2 For all $P_i, P_j$, $i, j \in \text{SYS}$, and $\forall t \geq 0$, $|C_i(t) - C_j(t)| \leq \epsilon$.

CA3 If $e$ and $e'$ are events in process $P_i$ that take place at times $t$ and $t'$ respectively, then $C_i(t) \neq C_i(t')$.

The global property detected by the algorithm is a conjunctive global predicate $\mathcal{A}$. Let $\mathcal{A}_i$ denote a predicate over the state of process $P_i$. Then, $\mathcal{A} = \wedge_i \mathcal{A}_i$, $i \in \text{SYS}$. The conjunctive global predicate becomes true when the local predicates of each process become true simultaneously.

### 3.2. The Algorithm

#### 3.2.1. Description

In this section, we present the algorithm for detection of unstable conjunctive global predicates. The algorithm is given as a set of events executed by process $P_i$ in response to events that occur when it is in a given state. Figure 3.1 contains the state information maintained, and functions supported, by process $P_i$. Figure 3.3 gives the algorithm. Execution of each statement in the algorithm is assumed to be atomic.
RecvToken():
if (p == i) then
    TSClean ← TFSmin; p ← -1;
if (TSMax ≠ -1) then
    ASSERT A; TSMax ← -1; TFSmin ← -1;
end
Remove I_i ∈ γ_i such that (ETIME(I_i) ≠ -1) ∧ ((ETIME(I_i) - ϵ ≤ TSMAX) ∨ (ETIME(I_i) ≤ TSClean));
if γ_i empty then
    TSClean ← clock_i; p ← -1; TSMAX ← -1; TFSmin ← TSClean
    send Token(p, TSMAX, TFSmin, TSClean) to next_i; RETURN;
else
    /* Find max(TS_i) for current set of overlapping intervals */
    if ((TSMAX ≠ -1) ∧ (STIME(HeadOf(γ_i)) > TSClean)) ∨ ((TSMAX == -1) ∧ (STIME(HeadOf(γ_i)) ≥ TSClean - ϵ)) then
        TSMAX ← STIME(HeadOf(γ_i));
        if (ETIME(HeadOf(γ_i)) == -1) then
            TFSmin ← ETIME(HeadOf(γ_i));
        end
        p ← i;
    else
        TFSmin ← ETIME(HeadOf(γ_i));
    end
    /* Find min(TF_i) for current set of overlapping intervals */
    if (ETIME(HeadOf(γ_i)) == -1) then
        T ← clock_i;
    else
        T ← ETIME(HeadOf(γ_i));
    end
    if (T < TFSmin) then
        TFSmin ← T;
    end
else
    /* Find min(TF_i) for past set of overlapping intervals */
    if ((p ≠ -1) ∧ (ETIME(HeadOf(γ_i)) ≠ -1) ∧ (ETIME(HeadOf(γ_i)) < TFSmin) ∨ (TFmin == -1) ∧ (TSClean ≠ -1) ∧ (ETIME(HeadOf(γ_i)) ≠ -1)) then
        TFSmin ← ETIME(HeadOf(γ_i)); p ← i;
    end
end
send Token(p, TSMAX, TFSmin, TSClean) to next_i; RETURN;

Figure 3.2: Actions taken at P_i on Receipt of Token
Each process $P_i$, $i \in SYS$, executes the same algorithm, with the exception that a unique, but arbitrary, process is designated to initiate a single detection token into the system. Process $P_i$ locally collects intervals, delimited by local clock readings, over which $A_i$ is true. A token circulates to find a set of intervals $\Sigma$, one from each process, that have sufficient overlap to ensure the predicate was true at an instant of real time. The overlap is required to account for the approximate clock synchrony. The token circulates continuously, so that intervals which cannot belong to a set of overlapping intervals are periodically removed.

The accumulation of intervals is triggered by changes in the truth value of the local predicate. When $A_i$ becomes true, an element $\langle TS_i, -1 \rangle$, where $TS_i$ is the value of $C_i$ when $A_i$ became true, is added to a queue $\gamma_i$ maintained by process $P_i$. Intervals are stored in $\gamma_i$ in the order in which they are generated. When $A_i$ becomes false, the element $\langle TS_i, -1 \rangle$ in $\gamma_i$ is modified to $\langle TS_i, TF_i \rangle$, where $TF_i$ is the value of $C_i$ when $A_i$ became false. Intervals in which $TF_i$ is negative are known as open intervals; other intervals are closed. The use of open intervals allows the detection of stable global predicates in addition to unstable ones. (Stable predicates, once true, remain true indefinitely [10].)

Processes are in one of two states: waiting or procTktn. $P_i$ transitions from waiting to procTktn on receipt of the detection token. On this transition, the routine recvToken(), given in Figure 3.2, is initiated and executes concurrently. While in the procTktn state, events related to changes in the truth of the local predicate

<table>
<thead>
<tr>
<th>state</th>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>waiting</td>
<td>$A_i$ becomes true</td>
<td>Insert($\gamma_i, \langle clock_i, -1 \rangle$)</td>
</tr>
<tr>
<td></td>
<td>$A_i$ becomes false</td>
<td>CloseInterval($\gamma_i$)</td>
</tr>
<tr>
<td></td>
<td>Token($p, TS_{max}, TF_{min}, TS_{clean}$) received</td>
<td>state_i ← procTktn; Initiate recvToken() processing</td>
</tr>
<tr>
<td>procTktn</td>
<td>$A_i$ becomes true</td>
<td>Insert($\gamma_i, \langle clock_i, -1 \rangle$)</td>
</tr>
<tr>
<td></td>
<td>$A_i$ becomes false</td>
<td>CloseInterval($\gamma_i$)</td>
</tr>
<tr>
<td></td>
<td>recvToken() processing completes</td>
<td>state_i ← waiting</td>
</tr>
</tbody>
</table>

Figure 3.3: Algorithm for Process $P_i$
take precedence over execution of the statements of `RecvToken()`. If the local predicate becomes true during execution of statement $x$ of `RecvToken()`, then the local interval queue will be modified according to the algorithm, and execution will return to statement $x + 1$ of `RecvToken()`.

### 3.2.2. Correctness

The following definitions and lemmas are useful in establishing the correctness of the algorithm.

**Definition 1.** Let $T_i$ be a clock value generated by the clock at $P_i$ and let $T_j$ be a clock value generated by the clock at $P_j$. If $i \neq j$, then $T_i <_{rt} T_j$ iff $T_i < T_j - \epsilon$. If $i = j$, $T_i <_{rt} T_j$ iff $T_i < T_j$.

The following lemma shows that when the relation $<_{rt}$ holds on clock values $T_i$ and $T_j$, then the associated real time instants can be ordered correspondingly, that is, when $T_i <_{rt} T_j$, then $C_i$ read $T_i$ before $C_j$ read $T_j$.

**Lemma 1.** Let $C_i(t_i) = T_i$ and let $C_j(t_j) = T_j$. If $T_i <_{rt} T_j$, then $t_i < t_j$.

**Proof:** When $i = j$, then the lemma holds by clock axiom CA1. If $i \neq j$, then by clock axiom CA2, when the clock at $P_i$ reads $T_i$, the clock at $P_j$ must be within $\epsilon$ of $T_i$ and then $C_j(t_i) \leq T_i + \epsilon$. By assumption $T_i < T_j - \epsilon$. Then $C_j(t_i) \leq T_i + \epsilon < T_j$. Hence, by clock axiom CA1, $t_i < t_j$.

**Definition 2.** The triple $\langle TS, TF, i \rangle$ denotes that the local predicate $A_i$ of process $P_i$ was true continuously over a real time interval $[ts, tf]$ where $ts < tf, C_i(ts) = TS$, and $C_i(tf) = TF$.

**Definition 3.** Let $\Sigma$ be a set of intervals, each of the form $\langle TS_i, TF_i, i \rangle$, for each $i \in SYS$. Then $\Sigma$ is a set of overlapping intervals if $\forall i, j \in SYS, TS_i <_{rt} TF_j$.

If a set of intervals is overlapping, then clearly there must be an instant of time when all the local predicates are true simultaneously.

**Lemma 2.** Let $\Sigma$ be a set of overlapping intervals. Then there exists a real time instant when $\bigwedge_i A_i, i \in SYS$, holds.

SIU 2004
A Distributed Algorithm For Unstable Global Predicate Evaluation With Approximately Synchronized Clocks

PROOF: This follows trivially from Definition 3 and Lemma 1.

In order to establish the correctness of the algorithm, we must show that if the global predicate is detected, then there was an instant of time when all local predicates were simultaneously true (Safety). We must also show that if the predicate is true, then it will be detected (Liveness).

**Theorem 1** (Safety). If $P_j$ asserts $A$, then there exists some real time instant $t$ for which $A_i$ is true for all $i$ in SYS.

PROOF: The conjunctive global predicate is true when a set $\Sigma$ of overlapping intervals, one from each process in the system, is found.

$P_i$ asserts the global predicate to be true only when it receives a token containing its own identifier. If the token identifier is $i$ then, by the algorithm, the time-stamp $T_{S_{max}}$ contained in the token was generated by $P_i$.

If the token has returned to $P_i$, then every other process $P_j$, $j \neq i$, has propagated the token. $P_j$ propagates the token, only if $T_S \leq T_{S_{max}} <_{rt} T_F$. Then by Lemma 2, the intervals, one from each process, overlap in real time and there was a real time instant when the global predicate held.

As clocks are approximately synchronized, we require the global predicate be true sufficiently long to generate a set of overlapping intervals. In Section 4, we establish the minimum duration a predicate must be globally true to be detected by comparing clock values. We also show that a set of overlapping intervals will be generated if a predicate is globally true over this interval.

**Lemma 3.** Let $\Sigma = \{ I_i : I_i = (T_{S_i}, T_{F_i}, i), T_{S_i} <_{rt} T_{F_j}, i,j \in SYS \}$. Then, if $I_i \in \Sigma$, $I_i$ will not be removed from $\gamma_i$ until $\Sigma$ is detected.

PROOF: Only receipt of $\text{Token}(p, T_{S_{max}}, T_{F_{min}}, T_{S_{clean}})$ will cause $P_i$ to remove intervals. An interval $I_i = (T_{S_i}, T_{F_i}, i)$ is removed only if $T_{F_i} \leq T_{S_{max}} + \epsilon$ or $T_{F_i} \leq T_{S_{clean}}$.

Initially $T_{S_{max}}$ and $T_{S_{clean}}$ are negative. Thus, no intervals are removed while the values of $T_{S_{max}}$ and $T_{S_{clean}}$ are negative. When $T_{S_{max}}$ has a non-negative value, it denotes a potential max($T_{S_i}$), $i \in SYS$, for the current set of intervals. Thus, if an interval $I_i$ has an end time-stamp $T_{F_i} \leq T_{S_{max}} + \epsilon$, it has ended too
early to belong to the current set of intervals. Since the intervals are stored in
the order they were generated, the interval $I_i$ cannot belong to any future set of
overlapping intervals if it does not belong to the current set. Hence, $I_i$ can be
removed.

Every set $\Sigma'$ of overlapping intervals is characterized by a pair of time-stamps
$\{TS', TF'\}$, such that $TS' = \max(TS_i)$, and $TF' = \min(TF_i)$, where $(TS_i,TF_i)
\in \Sigma'$. Once this set of overlapping intervals has been detected, $TS_{\text{clean}}$ is assigned
the value of $TF'$. Processes that subsequently receive the token remove intervals
with end-time-stamps $TF_i \leq TS_{\text{clean}}$. Since all intervals belonging to $\Sigma'$ had in-
tervals that satisfied the condition $TS_i < TF' - \epsilon'$, $\forall i \in SYS$, it trivially follows
that the next set of overlapping intervals will have intervals with start-time-stamps
$TS_i \geq TF' - \epsilon'$. Then, all closed intervals with end-time-stamps $TF_i \leq TF'$ can
be removed since they belong to some previous set of overlapping intervals and
cannot belong to some future set of overlapping intervals. When the token is re-
ceived by a process that has an empty interval queue, $TS_{\text{clean}}$ is assigned the value
of the local clock reading at the time the token is received. In subsequent recip-
ients, the closed intervals with end-time-stamps $TF_i \leq TS_{\text{clean}}$ can be removed
since the local predicate of a process (and thus the global predicate) was not true
until $TS_{\text{clean}}$ as read by some process.

Thus, the algorithm only removes intervals that cannot belong to the current or
a future set of overlapping intervals.

\textbf{Theorem 2 (Liveness).} Suppose a set $\Sigma$ of overlapping intervals is generated
by the processes in SYS. Then some process $P_k$, $k \in SYS$, will assert $A$.

\textbf{Proof:} By Lemma 3, an interval that belongs to an undetected set of overlap-
ning intervals is not removed from the interval queue of a process until the set
has been detected. Consider the case when a set $\Sigma''$ of overlapping intervals has
been detected. The token component $TS_{\text{clean}}$ is assigned the value of $TF_{\text{min}}$ and
the components $p$, $TS_{\text{max}}$, and $TF_{\text{min}}$ are set to $-1$. The initial condition can be
considered as a special case of an instance when the global predicate has been
asserted, with the value of $TS_{\text{clean}}$ as $-1$. In case $TF_{\text{min}}$ was the local clock read-
ing of a process and not the interval end time-stamp of a closed interval, then
the smallest closed interval end time-stamp $TF_m$ from $\Sigma'$, is located by passing the token around. Once this is found, the token circulates the virtual ring until it reaches a process with an interval that belongs to the next set $\Sigma'$ of overlapping intervals, in chronological order. When determining the next set of overlapping intervals, only those intervals with a start time-stamp $TS_i$ such that $TS_i \geq TS_{\text{clean}} - \epsilon$ are considered, since this condition eliminates the possibility of a set of overlapping intervals being detected more than once. In case $TS_{\text{max}}$ and $TF_{\text{min}}$ belonged to the same process, the interval with start time-stamp $TS_{\text{max}}$ would be removed once the detection is complete. Due to this, any interval in that process considered for a subsequent detection would have a start time-stamp $T' \geq TS_{\text{clean}} - \epsilon$ (and hence we do not need to check if $T' \geq TS_{\text{clean}}$). Before reaching such an interval with start time-stamp $T \geq TS_{\text{clean}} - \epsilon$, intervals that ended too early to be in the set of overlapping intervals being currently evaluated or those that belonged to some previous set of overlapped intervals are removed. After the first interval that belongs to $\Sigma'$ has been located, the interval with start time-stamp $TS_m = \max(TS_i)$, $TS_i \geq TS_{\text{clean}} - \epsilon$ and $TS_i \in \Sigma'$, is located. Once the token has been time-stamped with $TS_m$ by process $P_k$, it would complete one trip around the ring and return to $P_k$, since every processes would have an interval $\langle TS_i, TF_i \rangle$ at the head of its queue such that $TS_i \leq TS_m < rt TF_j$, $\forall i, j \in \text{SYS}$. The detection of the set of overlapping intervals is then complete, and the global predicate is asserted by process $P_k$. The evaluation starts afresh. Since the intervals are stored and evaluated in the order they were generated, the instances of overlapping intervals are also detected sequentially in the order of generation. This ensures that no instance of overlapping intervals goes undetected.

Thus, the algorithm detects all instances of overlapping intervals $\langle TS_i, TF_i, i \rangle$, $i \in \text{SYS}$, such that $TS_i < rt TF_j$, $\forall i, j \in \text{SYS}$. 

3.2.3. Performance

The proposed algorithm is more scalable, in terms of per process space and computation, than existing centralized algorithms. Let the maximum number of intervals generated by a process be $E$. The space required in each process in the proposed algorithm is $O(E)$, independent of the number of processes. The space
requirement in centralized algorithms is $O(EN)$, where $N$ is the number of system processes. The number of intervals in any process queue can be reduced further by discarding intervals of duration lesser than or equal to $2\epsilon$, since they cannot contribute to any set of overlapping intervals (see section 4).

Existing algorithms that are centralized have computational complexity of $O(EN\lg N)$ [4] and $O(EN^2)$ [6]. The algorithm presented in this paper requires computation at each process that is $O(NE)$ in the worst case, when none of the intervals generated by processes overlap.

The message passing complexity of our algorithm is $O(EN^2)$. Centralized approaches have message passing complexity of $O(EN)$. The time required to detect the predicate once it becomes true is $O(EN\delta)$, where $\delta$ is the maximum message delivery delay. The worst case delay is experienced when the intervals generated by successive processes each have potential largest start time-stamps and the time between generation of successive intervals is less than the time for the token to circulate completely through the cycle of processes.

The detection delay can be reduced through initiation of a new token for every interval generated by a process. The required modifications to the algorithm include the following.

(i) A process receiving a token should discard it if no interval overlaps with the interval indicated in the token, or the process has an overlapping interval with greater start time-stamp. As before, the token with the largest start timestamp for a set of overlapping intervals will detect the global predicate.

(ii) A clean-up token should be initiated by the process that detects the global predicate.

The detection time of this modified algorithm is $O(ND\delta)$, where $D$ is the maximum token processing time at each process. The message passing complexity of this algorithm is $O(EN^2)$. The design of the proposed algorithm ensures that intervals that can no longer contribute to any overlapping intervals are removed over time by the detection token.
4. Lower Bound on Predicate Duration

In this section we show that, to ensure detection, the global predicate must be true for a real time interval of sufficient duration that each process clock passes through an interval greater than $2\epsilon$. If the global predicate is true for a shorter duration, it may not be detected.

The following lemmas are useful in presentation of the proofs establishing the lower bound. The first result establishes that if the starting time-stamp of an interval at $P_i$ is not less than the ending time-stamp of an interval in $P_j$ by more than $\epsilon$, there need not exist an instant of time when the intervals overlapped.

**Lemma 4.** Suppose $\langle TS_i, TF_i, i \rangle$ and $\langle TS_j, TF_j, j \rangle$, $i \neq j$. If $(TS_j \geq TF_i - \epsilon) \lor (TS_i \geq TF_j - \epsilon)$, then the interval in which $A_i$ is true and the interval in which $A_j$ is true may not overlap at an instant of time.

**Proof:** In order for two intervals to overlap in real time, each interval must start before the other ends (in real time). Let $tf_i$ be the instant when the interval at $P_i$ ends and let $ts_j$ be the instant when the interval at $P_j$ begins. By clock axiom CA2, the clock at $C_j$ must read a value within $\epsilon$ of $TF_i$ at $tf_i$, that is, $TF_i - \epsilon \leq C_j(tf_i) \leq TF_i + \epsilon$. Suppose then that $C_j(tf_i) = TF_i - \epsilon$ and that $TS_j = TF_i - \epsilon$. Then $C_j(tf_i) = C_j(ts_j) = TS_j$ and by the clock axioms, it is possible that $tf_i < sup(C_j^{-1}(TS_j))$ and $ts_j = sup(C_j^{-1}(TS_j))$.

From Lemma 1, it is clear that if for a set of intervals $\{\langle TS_i, TF_i, i \rangle, i \in SYS\}$, it is true that $TS_i \prec TF_j$, $i \neq j$, $i, j \in SYS$, then the intervals overlap in real time. The following theorem then follows directly from this observation and Lemma 4.

**Theorem 3.** The intervals $\{\langle TS_i, TF_i, i \rangle \mid i \in SYS\}$ are known to overlap only if $(TS_j \prec TF_i)[i, j \in SYS, i \neq j]$.

We now show that each clock must pass through an interval of $2\epsilon$ while the predicate is globally true to ensure that intervals whose time-stamps adhere to the requirements of Theorem 3 are generated. We first show that global truth for greater than $2\epsilon$, as read by any process clock, will produce intervals $\langle TS_i, TF_i, i \rangle$, for all $i$ in $SYS$, that meet the requirements of Theorem 3. We then show that if the
conjunctive global predicate is true for less than or equal to $2\epsilon$, intervals satisfying
Theorem 3 may not be generated.

**Lemma 5.** Suppose $A_i$ is true over the real-time interval $[ts_i, tf_i]$, $\forall i \in SYS$, and that $ts_i \leq ts < tf \leq tf_i$, where $C_i(tf) - C_i(ts) > 2\epsilon$, $\forall i \in SYS$. Then $\forall i, j \in SYS, TS_i < rt, TF_j$, where $TS_i = C_i(ts_i)$ and $TF_j = C_j(tf_j)$.

**PROOF:** The event which turns a local predicate from false to true must be different from the event which turns a local predicate from true to false. Then by clock axiom CA3, $TS_i \leq C_i(ts) < C_i(tf) \leq TF_i$ and $TS_j \leq C_j(ts) < C_j(tf) \leq TF_j$.

By assumption, the local predicate is true at $P_i$ for an interval greater than $2\epsilon$ as read by $C_i$; then $C_i(ts) + 2\epsilon < C_i(tf)$ and $TS_i < rt, TF_j$. We can similarly show that $TS_i < TF_j - \epsilon$.

**Lemma 6.** Suppose $A_i$ is true over the real-time interval $[ts, tf]$ and $C_i(tf) - C_i(ts) \leq 2\epsilon$, $\forall i \in SYS$. Then $C_i(ts) < rt, C_j(tf), i \neq j, \forall i, j \in SYS$, may not hold.

**PROOF:** By assumption, $C_i(tf) \leq C_i(ts) + 2\epsilon$. From clock axiom CA2, $C_i(ts) - \epsilon \leq C_j(ts) \leq C_i(ts) + \epsilon, \forall i \in SYS$. Suppose that $C_i(tf) = C_i(ts) + 2\epsilon$. Then $C_j(ts) \leq C_i(tf), \forall j \in SYS$.

Thus, if the conjunctive global predicate is true for lesser than or equal to $2\epsilon$, intervals for which $TS_i < TF_j - \epsilon$ and $TS_j < TF_i - \epsilon$ may not be produced. The following theorem, that follows directly from Lemmas 5 and 6, states this formally.

**Theorem 4.** Suppose $A_i$ is true over the interval $[ts, tf], \forall i \in SYS$. Then always $TS_i < rt, TF_j, i \neq j, i, j \in SYS$, only if $C_i(ts) + 2\epsilon < C_i(tf)$.

Finally, we establish the lower bound.

**Theorem 5.** The conjunctive global predicate must remain true globally for more than $2\epsilon$, as read by any process clock, for certain detection.
PROOF: By Theorem 3, in order to assert that a set of intervals \( \{TS_i, TF_i, i \mid i \in SYS\} \) overlap in real time, the condition \( TS_i <_{rt} TF_j, i \neq j, i, j \in SYS \) must hold. By Theorem 4, time-stamps that have these properties are certain to be generated only when the predicate is globally true for more than \( 2\epsilon \), as read by \( C_i \), for all \( i \) in \( SYS \).

5. Summary

Evaluation of unstable global predicates finds several applications in distributed computing. The problem is complicated by the lack of a common clock and common memory. Algorithms proposed in the past use causal ordering to find consistent global states in which the global predicate could have held true in a computation. However, causal ordering cannot always be used to detect if a global predicate held true at some instant in time during a computation without freezing the underlying computation. Recent efforts toward this problem use time provided by approximately synchronized clocks in centralized and distributed algorithms. Centralized algorithms, in general, are not conducive to scaling. They impose high space and computation overheads in the central process. In this paper we presented an algorithm for the detection of unstable conjunctive global predicates in systems with approximately synchronized clocks that is designed to be more scalable. We also established the lower bound on the duration a predicate must remain true globally, in order to be detected in environments with approximately synchronized clocks.

References


Authors addresses:
Shilpa Bansod.
Network Associates
3965 Freedom Circle, Santa Clara, CA 95054
Shilpa_Bansod@NAI.com
Jean Mayo.
Department of Computer Science
Michigan Technological University
Houghton
MI 49931
jmayo@mtu.edu