# Resettable Vector Clocks[1]

*Anish Arora*† *Sandeep S. Kulkarni*‡ *Murat Demirbas*†

† Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA

‡ Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA

[1]Email: {anish,demirbas}@cis.ohio-state.edu, sandeep@cse.msu.edu ;
Contact author is Murat Demirbas. Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ;

This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, a grant from Microsoft Research, NSF CAREER CCR-0092724, DARPA contract F33615-01-C-1901, and ONR Grant N00014-01-1-0744.

Preliminary results from this paper (with the exception of Section 7) appeared in [3].

**Abstract**

Vector clocks (VC) are an inherent component of a rich class of distributed applications. In this paper, we first consider the problem of realistic implementation —more specifically, bounded-space and fault-tolerant— of applications that use vector clocks (VC). To this end, we generalize the notion of VC to resettable vector clocks (RVC), and provide a realistic implementation of RVC. Further, we identify an interface contract under which our RVC implementation can be substituted for VC in client applications, without affecting the client's correctness. Based on such substitution, we show how to transform the client so that it is itself realistically implemented; we illustrate our method in the context of Ricart-Agrawala's mutual exclusion program and Garg-Chase's predicate detection program.

**Keywords :** Vector Clocks, Reset, Component Substitutability, Bounded-Space, Fault-Tolerance, Self-Stabilization.

# 1 Introduction

Distributed systems deal with the lack of a global time by using other abstractions of time. The causality relationship among events is a powerful concept that provides one such abstraction for reasoning about events in a distributed computation. Fidge [11] and Mattern [15] independently proposed vector clocks that capture the causality relation between events. Vector clocks (VC) are extensively used in distributed applications, such as, distributed debugging [12], checkpointing and recovery, and causal communication [6, 20].

Applications that use VC may require unbounded space since vector clocks grow unboundedly. To realistically implement VC applications using bounded space, VC should also be bounded. We observe that many applications are structured in phases and track causality only within a bounded number of adjacent phases. This suggests that we can reuse timestamps by augmenting VC with a "nonblocking reset" operation and enabling a process to locally reset its own local clock whenever it moves from one phase to another. By using a resettable vector clock component (RVC) instead of VC, a bounded number of clock values suffice provided that no two timestamps of different incarnations coexist.

Fault-tolerance is another essential ingredient of realistic implementation of applications. As such, substituting VC with bounded-space RVC is not enough; RVC should also be fault-tolerant. The issues of boundedness and fault-tolerance are interrelated. One might argue that using a 64-bit register for a clock value should suffice for bounding most applications; in the presence of faults however this argument does not hold. E.g., if clocks are improperly initialized or lose coordination, they may reach their maximum value quickly, or simultaneous occurrence of process crash and message reorder can drive applications into arbitrary states, including ones where all space is used up. Hence, for an RVC application to tolerate such faults, it is necessary that RVC enjoy some fault-tolerance properties.

In this paper, we consider two problems related to realistic implementation of VC applications. The first problem (which we call the RVC problem) deals with how to substitute

VC with a bounded-space and fault-tolerant RVC without affecting application correctness. Since not every application that uses VC can instead use bounded-space RVC, a related issue is to find a contract that an application needs to satisfy in order to permit this substitution. Our goals in solving the RVC problem are as follows. The identified contract should be minimal, so many applications will naturally satisfy it. Verification of whether an application satisfies the contract should be easy. Also for the sake of efficiency, the RVC implementation should not generate extra messages or block the application in the absence of faults. And lastly, in the presence of faults, the RVC implementation should recover from arbitrary states to states from where it correctly tracks the causality relation; this sort of fault-tolerance is commonly referred to as self-stabilization [8].

The second problem deals with how to transform VC applications into fault-tolerant ones, using RVC component as a building block. To this end, we use a method that is based on the following observation: In any application that uses a component, faults occur at three levels: (1) in the component, (2) in the application, or (3) in the interface between the component and the application. Our method deals with these three levels separately. By substituting a self-stabilizing RVC component with VC, we deal with faults at level 1. By exporting a reset interface for RVC, we deal with faults at level 3.

**Contributions of the paper.** We give a bounded-space implementation of the VC component, and show how to make it self-stabilizing. Further, we show that any application that uses VC can instead use our implementation without loss of correctness, provided that the application satisfies a two-part contract. Each part gives a lower bound on minimum size of RVC; the size of our implementation matches the maximum of these two bounds. The first part identifies the temporal localities within which the application tracks event causality; the bound on clock size is directly proportional to the size of these temporal localities. The second part identifies how often processes communicate; the bound is inversely proportional to the frequency of communication.

We use Ricart and Agrawala's mutual exclusion [19] as a running example and thusly demonstrate our method for transforming a VC-based application to a bounded-space, self-stabilizing version thereof. To the best of our knowledge, prior implementations for Ricart-Agrawala's mutual exclusion have lacked bounded-space and stabilizing tolerance.

Finally, as an application of our method, we present a transient predicate detector based on Garg and Chase's weak conjunctive predicate detector program [12]. Our detector improves upon that in [12, 13] since our detector is online, stabilizing-tolerant and has bounded implementation.

**Related work.** In [22], the necessary and sufficient condition for a correct clock resetting is establishes and a clock reset protocol is presented. Their reset rule asserts that messages must not cross any reset line. By using a contract, we are able to relax that rule, and use a "reset window" instead of a "reset line". The main difference in our works is that in [22], the VC property holds only within each phase, whereas in our case VC property holds even across phases. In [22], when the clock of one process is reset, this triggers a network wide reset of clocks, introducing extra causality among phases. Since their protocol flushes all messages between phases, there is no possibility to compare two timestamps that are from different phases. Whereas in our case the clock of each process can be reset independently without incurring a reset operation on other processes, therefore we are able to compare across phases and find concurrent events. That is, our protocol does not introduce extra causalities.

Mostefaoui and Theel [18] have presented a solution that reduces the size of the vector clock. In their solution, whenever any clock reaches a predetermined limit, the application is blocked and the clocks of all processes are reset to zero. An unbounded variable *phase* is maintained to count the number of resets that have been performed. By maintaining a single phase, the size of the vector clock is reduced. However, because additional messages are sent which change the clock values, the causality relation in the underlying application is changed. Other work [10, 16, 21] also has addressed reducing the size of vector clocks and the overhead associated with piggybacking timestamps to the application messages. These solutions are neither bounded-space nor stabilizing fault-tolerant. By way of contrast, our solution achieves both.

Awerbuch, Patt-Shamir and Varghese [5] have addressed the problem of bounding the registers used in network protocols. In their approach, reaching the bound is considered to be a fault and, therefore, a global blocking reset operation is invoked to reset the application. By way of contrast, we allow nonblocking resets to be performed by a process when it reaches the bound of its local clock, without treating this scenario as a fault or

affecting the application correctness. Furthermore we enable application stabilization via RVC stabilization, whereas in [5] the application is assumed to be stabilizing and transforming the registers to be bounded is shown to preserve application stabilization.

In contrast to our mutual exclusion program, the original version due to Ricart and Agrawala [19] uses $O(log\ n)$ space per process, where $n$ is the number of processes, and is not stabilizing fault-tolerant. Mizuno et. al. [17] gave a stabilizing version of the Ricart-Agrawala program, but their solution uses unbounded state. Our solution uses $O(n\ \ log\ n)$ space per process.

In contrast to our transient predicate detection program, the Garg and Chase predicate detector [12] is not online, bounded space or self-stabilizing. Afek and Dolev also present a transient predicate detector in [1]. However, their predicate detector assumes a synchronized execution model, whereas ours does not.

**Organization of this paper.** In Section 2, we present the system model, preliminary definitions and our notation. We formally define the RVC problem in Section 3. In Section 4, we specify the contract, and present our bounded-space implementation of RVC. We show how this implementation is made self-stabilizing to transient faults in Section 5. In Section 6, we use the bounded-space and fault-tolerant RVC to provide a bounded and self-stabilizing implementation of the mutual exclusion solution by Ricart and Agrawala [19]. Also, in Section 7, we present an online, stabilizing, and bounded transient predicate detector as an application of our method. Finally, we make concluding remarks in Section 8.

# 2 Preliminaries

In this section, we discuss the system and fault model and introduce the notations to be used in the rest of the paper.

**System model.** A program consists of a set of processes which communicate via message passing on interprocess channels. Execution is asynchronous, i.e., every process executes at its own speed and messages in the channels are subject to arbitrary but finite transmission delays. We assume that the processes are connected but we do not assume

that they should be fully connected. We assume that the channels are of bounded size, but we do not assume the channels to be FIFO or bidirectional (unless the application requires so).

Process execution consists of a sequence of events. Each event is the execution of some process action and is in one of three forms: local event, message send event, or message receive event. An action is an atomic and terminating statement that updates 0 or more program variables and calls 0 or more component operations. Lamport's [14] happened before (causality) relation $\underline{hb}$, induces a partial order on the set of events; it is the smallest transitive relation that satisfies the following two conditions:

- if events $e$ and $f$ are in the same process and $e$ occurred before $f$, then $e \ \underline{hb} \ f$
- if $e$ is a sending of a message $m$ and $f$ is the receipt of $m$, then $e \ \underline{hb} \ f$.

Events $e$ and $f$ are concurrent, denoted as $e \ \underline{co} \ f$, iff both $e \ \underline{hb} \ f$ and $f \ \underline{hb} \ e$ are false, i.e., $e \ \underline{co} \ f \quad \equiv \quad \neg(e \ \underline{hb} \ f) \wedge \neg(f \ \underline{hb} \ e)$ .

Semantically, a *program* is defined by its set of computations. A *computation* is an alternating sequence of states and events whose projection on the events extends the causality relation. A state gives a value for each variable (chosen from its respective domain), and the sequence of messages in each channel.

**Faults.** In our model, messages can be corrupted, lost, or duplicated at any time. Moreover, processes (respectively channels) can be improperly initialized, fail, recover, or their state could be transiently (and arbitrarily) corrupted at any time. For ease of exposition, we assume the number of fault occurrences is finite.

**Self-Stabilization.** A program $P$ is *self-stabilizing* iff starting from an arbitrary state $P$ eventually recovers to a state from where its specification is satisfied.

**Component.** A component consists of an interface and an implementation. The interface consists of a nonempty set of values and a set of operation names. Associated with each operation is a list of arguments. The implementation consists of an operation body for each operation. When an operation is invoked, its body is executed atomically and always terminates.

**Notation.** In this paper, we use $i, j$, and $k$ to denote processes. We use $e$ and $f$ to denote events. Where needed, events are subscripted with the process at which they occur, thus, $e_j$ is an event at $j$. We use $m$ to denote messages. Messages are subscripted by the sender process, thus, $m_j$ is a message sent by $j$.

A formula $(op\ e, f : R.e.f : X.e.f)$ denotes the value obtained by performing the (commutative and associative) $op$ on the $X.e.f$ values for all $e, f$ that satisfy $R.e.f$. As special cases, where $op$ is conjunction, we write $(\forall e, f : R.e.f : X.e.f)$, and where $op$ is disjunction, we write $(\exists e, f : R.e.f : X.e.f)$. Thus, $(\forall e, f : R.e.f : X.e.f)$ may be read as "if $R.e.f$ is true then so is $X.e.f$", and $(\exists e, f : R.e.f : X.e.f)$ may be read as "there exists $e$ and $f$ such that both $R.e.f$ and $X.e.f$ are true". Where $R.e.f$ is true, we omit $R.e.f$. If $X$ is a statement then $(\forall e, f : R.e.f : X.e.f)$ denotes that $X$ is executed for all $e$, $f$ that satisfy $R.e.f$. This notation is adopted from [9].

Finally, we denote a variable $var$ at process $j$ as $var.j$ . If $var.j$ is an array, e.g. which contains an entry for all processes, we use $var.j[k]$ to denote the $k^{th}$ entry in $var.j$ .

# 3 The Problem of Resettable Vector Clocks

In this section, we formally define the Resettable Vector Clocks problem. Towards this end, we first present the interface of VC and then show how this interface is extended to obtain the interface of RVC. Subsequently, we formally define what it means for RVC to be substitutable for VC. To illustrate the definitions, we use a variation of Ricart-Agrawala mutual exclusion that uses VC, namely RA, as a running example.

**RVC interface.** The interface of RVC subsumes that of VC so we begin with VC. The values in the interface consists of a vector; each element in the vector is a timestamp. The operations in the interface of VC are as follows:

- `constructor( process_id_list)`
- `send( sender_id, event, message, flag )`
- `receive( receiver_id, event, message, flag )`

- `local-event( process_id, event, flag )`
- `happened-before( event1, event2 ) : boolean`
- `concurrent ( event1, event2 ) : boolean` [2]

For the reader's convenience we give an implementation of VC in the Appendix. Note that the above interface is more general than the traditional one in that, a boolean argument *flag* is associated with the `send`, `receive`, and `local-event` operations to indicate whether or not a fresh timestamp is to be given to the event as opposed to the current timestamp. The intuition behind this generalization is that applications that use VC compare only certain events of interest, and only these events of interest need to be freshly timestamped rather than all the events. For example, in the case of causal broadcast, the only events whose timestamps are compared are the send events, and only these need to be freshly timestamped; the receive or local events do not need to be freshly timestamped.

The interface of RVC modifies the constructor to include a client contract (see Section 4.1) as an extra argument, it also includes these additional operations:

- `nonblocking-reset( process_id ):` This operation resets the local clock value of the given process without introducing extra communication messages or blocking. Other processes learn about this reset gradually.
- `global-reset( ):` This operation performs a blocking reset for all processes and introduces extra communication messages. This operation is not executed in the absence of faults, and invoked only upon the detection of a fault.

Let $P$ be a program and $D$ be a subset of the process actions of $P$.

**Definition.** $P$ is *a well-formed client of* VC (RVC) iff (1) $P$ calls the send (respectively, receive) operation of VC (RVC) whenever it sends (respectively, receives) a message, and (2) all the events that $P$ compares by the `happened-before` operation of VC (RVC) have been freshly timestamped by calling the `local-event`, `send` or `receive` operations of VC (RVC) with $flag = True$. □

[2] *Remark.* Since the implementation of the "concurrent" operation is straightforward (i.e. $(e \underline{co} f) \equiv (\neg(e \underline{hb} f) \wedge \neg(f \underline{hb} e))$) we will not discuss its implementation any further in this paper.

To illustrate the above definition, we now present our case study, the mutual exclusion algorithm by Ricart Agrawala [19]. Our illustration shows that RA is a well-formed client of VC.

---

**Case Study.** In RA, whenever $j$ wants to enter the *critical section*, CS, it sends a timestamped *REQUEST* message to all the processes. When a process $k$ receives a *REQUEST* message from $j$, it defers the *REPLY* message to $j$ iff $k$ has requested CS with a lower timestamp than $j$'s request. Otherwise, $k$ sends a *REPLY* message to $j$. $j$ enters the CS after it has received *REPLY* messages from all other processes. When $j$ exits CS, it sends all the deferred *REPLY* messages.

In RA, $j$ maintains the following variables.

- *REQ_ts.j*: the timestamp of the last *REQUEST* at $j$.
- *deferred_set.j*: the set of processes that $j$ has to send a *REPLY* when it releases the CS.
- *REPLY.j.k*: $j$'s *REPLY* to $k$.
- *RECVD.j.k*: last *REPLY* $j$ received from $k$.
- *CS.j*: a boolean denoting whether or not $j$ is currently accessing CS.
- *hungry.j*: a boolean which is true iff $j$ is requesting for CS and has not completed its CS.

RA uses the *process-id*s to induce a total order over the events it compares. We implement this as follows:

```
less-than ( e_j , f_k ) : boolean
   return ( happened-before( e_j , f_k ) ∨
            (concurrent( e_j , f_k ) ∧ (j < k )));
```

Since RA compares only *Request CS* events, only these events are freshly timestamped by calling the `local-event` operation with *flag = True*. For all the other events, since we do not need fresh timestamps, we call the corresponding operations with *flag = False*. We denote these dummy events with the symbol "*". Therefore, the resulting process actions for $j$ are as follows.

**RA**

{ *Request CS* } $\longrightarrow$
    if ($\neg hungry.j$) then
        **local-event**($j$ , $REQ\_ts.j$ , *True*);
        $hungry.j$:=*True*;
        ($\forall k : k \neq j$ : **send**($j$ , * , $REQ\_ts.j$ , *False*));

{ *Receipt of* $REQ\_ts.k$ } $\longrightarrow$
    **receive**($j$ , * , $REQ\_ts.k$ , *False*);
    $REPLY.j.k := REQ\_ts.k$;
    if ($hungry.j \wedge \neg$**less-than** ($REQ\_ts.k$ , $REQ\_ts.j$))
    then $deferred\_set.j := deferred\_set.j \cup \{k\}$;
    else **send**($j$ , * , $REPLY.j.k$ , *False*);

{ *Receipt of* $REPLY.k$ } $\longrightarrow$
    $RECVD.j.k := REPLY.k$;

$hungry.j \wedge \forall k : k \neq j : REQ\_ts.j = RECVD.j.k \longrightarrow$
    $CS.j := True$;

{*Release CS* } $\longrightarrow$
    ($\forall k : k \in deferred\_set.j$:
                **send**($j$ , * , $REPLY.j.k$ , *False*));
    $deferred\_set.j := \emptyset$;
    $hungry.j := False$;
    $CS.j := False$

Note that RA is a well-formed client of VC.

---

**Definition.** *P reset annotated at D*, denoted as $P_D$, is the program $P$ with each action $ac$ of any process $j$, $ac \in D$, modified to append the call `nonblocking-reset`($j$) to the statement of $ac$. □

Note that, if $P$ is a well-formed client of VC then $P_D$ is a well-formed client of RVC. Given a program $P_D$, we refer to the actions of $D$ as distinguished actions of $P_D$, and any execution of a distinguished action of $P_D$ as a distinguished event of $P_D$.

---

**Case study (continued).** In RA, the processes loop through *Request CS*, *CS granted* (action 4), and *Release CS*. Since each loop execution can be seen as a new phase, we can select any of these three actions as the distinguished action of RA. We identify the *Release CS* as the distinguished action of RA, and append it with a call to `nonblocking-reset`

operation for $j$. We denote the resulting program, which is a reset annotation of RA at the *Release CS* action, as $\text{RA}_D$. Note that $\text{RA}_D$ is a well-formed client of RVC.

---

**Definition.** Let $P$ be a well-formed client of VC. RVC *is substitutable for* VC *in* $P$ iff there exists a set $D$ such that every event in $P_D$ that is an execution of `happened-before` returns the same result as the corresponding event in $P$. □

Intuitively, the definition states that RVC is substitutable for VC in $P$ iff the set of computations (i.e., the correctness) of $P$ is not affected by that substitution.

**The RVC problem.** *Design a bounded-space and stabilizing fault-tolerant RVC implementation and identify a contract $C$ such that, for every $P$ that is a well-formed client of VC and that satisfies $C$, RVC is substitutable for VC in $P$.* □

Note that while the client program accesses RVC, RVC is not allowed to access (the state of) the client program. Secondly, from the definition of substitutability and reset annotation, our transformation disallows any changes to the nature of the client program, such as the introduction of extra causalities in the absence of faults. Without having this second condition, one might consider a trivial solution to our substitutability problem such as: "RVC blocks the client program when the clock entry for a process is filled up and resets this clock entry on all the processes". This is not an acceptable solution because it introduces extra causalities which were not present in the original program, and the definition for substitutability is violated, since the happened-before operation for the blocking RVC does not return the same result as that of VC for all the events that any client can compare.

# 4 Bounded-Space Resettable Vector Clocks

In this section, we present a bounded implementation of RVC and identify a contract that should be satisfied for the clients to use this component; the contract consists of two parts: 1) a comparison predicate over the events whose causality needs to be tracked, and 2) a communication pattern between the processes. Finally, we present the main

theorem that identifies the conditions under which bounded RVC can be substituted for VC.

## 4.1 Contract

**Need for a comparison predicate.** Consider an application that needs to compare any two events in its computation. Since the total number of events may be unbounded, the application cannot use bounded vector clocks. In other words, the application must provide some predicate such that it will need to compare two events $e$ and $f$ only if the predicate holds for $e$ and $f$.

**Need for a communication pattern.** In order to provide a bounded implementation of vector clocks, it is necessary that processes communicate often enough. Consider a scenario where $e$ is a send event at $j$ and $f$ is the corresponding receive event at $k$; clearly, $e \ \underline{hb} \ f$. If $j$ performs a sufficiently large number of resets without ever communicating with $k$ then there exists an event $e'$ at $j$ such that the timestamps associated with $e$ and $e'$ are the same. Note that $e'$ is concurrent with $f$. As far as $k$ is concerned it cannot differentiate between $e$ and $e'$. Therefore, if $k$ receives a message with timestamp $e'$, it will update its clock incorrectly.

The problem with this scenario is that $k$ is not aware of an unbounded number of resets done by $j$. Therefore, for an application to use a bounded implementation of vector clocks, it must satisfy a communication pattern that guarantees the number of resets of $j$ that $k$ is not aware of is bounded. Likewise, it is also necessary that the number of reset events that a process can perform while a message is in transit is bounded; if a message $m$ stays in a channel for a long enough time such that a message $m'$ with identical timestamp is created, then the receiving process cannot update its clock appropriately.

From the above discussion, it follows that the contract should consist of a comparison predicate part and a communication pattern part. We discuss these two parts next.

### 4.1.1 Comparison predicate

Let $e_j$ and $f_k$ be events in some computation of $P_D$.

**Definition.** $PR^m.e_j$ denotes the $m^{th}$ distinguished event before $e_j$ at process $j$. $NX^n.e_j$ denotes the $n^{th}$ distinguished event after $e_j$ at $j$.

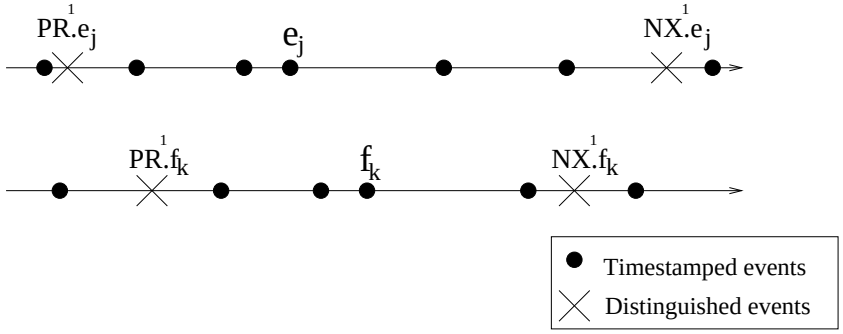


Figure 1: Illustration of $PR^1$ and $NX^1$ for events $e_j$ and $f_k$

For each process $j$ we insert an auxiliary distinguished event $R0_j$ before the beginning of the computation at $j$. We define $(\forall j, m : m \in Z^+ : PR^m.R0_j = R0_j)$ and $(\forall j, k :: R0_j \ \underline{hb} \ f_k)$ to be true for any non-auxiliary event $f_k$. For $m, n \in Z^+$, we define the comparison predicate $R(m, n)$ as follows:

$$R(m,n).e_j.f_k = (PR^m.e_j \ \ \underline{hb} \ \ f_k \ \ \wedge \ \ \neg(NX^n.e_j \ \ \underline{hb} \ \ f_k))$$

The client program can choose suitable values for $m$ and $n$ depending upon the comparison relation that is satisfied between events whose causality needs to be tracked.

**Definition.** A program $P_D$ satisfies the comparison predicate $R(m, n)$ for some $m, n \in N$ iff in every computation of $P_D$, $R(m, n).e_1.e_2$ holds between any two events $e_1$ and $e_2$ that $P_D$ compares by calling "happened-before($e_1$, $e_2$)" operation. □

*Observations.* (1) If a program satisfies $R(m, n)$, then it also satisfies $R(m', n')$ for any $m' \geq m$ and $n' \geq n$. This allows one to use conservative estimates for $m, n$ when it is difficult or impossible to determine the exact comparison predicate $R(m, n)$ that the application satisfies. (2) For any two events $e_1$ and $e_2$, "concurrent ($e_1$, $e_2$)" results in a call to happened-before($e_1$, $e_2$) and happened-before($e_2$, $e_1$). Hence, $P_D$ needs to

guarantee that both $R(m,n).e_1.e_2$ and $R(m,n).e_2.e_1$ holds. (3) $R(\infty,\infty)$ is equal to true. □

Many applications that use VC indeed satisfy $R(m,n)$ for some value of $m$ and $n$. For example, in fair mutual exclusion, processes compare their requests only with a bounded number of requests of other processes, or in checkpointing, processes need to track causality only between two consistent checkpoints. The values of $m$ and $n$ will depend upon how often processes communicate; the smaller the period for synchronization the smaller the $m$ and $n$ values. Thus, there is a trade-off between the clock size and synchronization frequency.

---

**Case study (continued).** To illustrate the use of the comparison predicate, we show that $\text{RA}_D$ satisfies the comparison predicate with $m=2$ and $n=1$.

**Lemma 1.** $\text{RA}_D$ satisfies $R(2,1)$.

**Proof.** For the purpose of proof, let's assume an unbounded auxiliary variable, $ph'$ such that $ph'.k.j$ denotes the number of distinguished events of $j$ that happened before the current event at $k$. Recall that RA compares only *Request CS* events. Let $e_j$ be the event denoting the request of $j$ for CS. Observe that $ph'.e_j.j$ denotes the number of critical sections $j$ has completed when it sent the request. When $j$ entered the critical section $ph'.e_j.j^{th}$ time, it obtained a reply from $k$. The value of $ph'$ in the request message $j$ sent for entering the critical section $ph'.e_j.j^{th}$ time, is $ph'.e_j.j-1$. Therefore, for any *Request CS* event $f_k$, $ph'.f_k.j$ is at least $(ph'.e_j.j-1)$. Also, $ph'.f_k.j$ cannot be greater than $ph'.e_j.j$ as it would imply that $k$ believes $j$ completed more critical sections than $j$ has actually completed.

Combining these two results, we have $ph'.e_j.j - 1 \leq ph'.f_k.j \leq ph'.e_j.j$ for any two events compared by RA. From the definition of $R$, $\text{RA}_D$ satisfies $R(2,1)$. □

---

### 4.1.2 Communication pattern

Consider the following communication pattern for $P_D$, where $M \in Z^+$.

$$
\begin{array}{l}
(\forall k, j, x : x \in N : \\
\quad (\exists e_k : (x^{th} \text{ distinguished event of } j \ \ \underline{hb}\, e_k) \ \wedge \\
\qquad \neg(x + M^{th} \text{ distinguished event of } j \ \ \underline{hb}\, e_k)) \\
\wedge\, (\forall m_k :: (\neg(x^{th} \text{ distinguished event of } j \ \ \underline{hb}\, \text{Send}(m_k)) \\
\qquad\qquad \Rightarrow \ (\text{Receive}(m_k) \ \ \underline{hb}\, x + M^{th} \text{ distinguished event of } j)
\end{array}
$$

This pattern states that, in any time-frame of $M$ distinguished events of $j$, all the processes deliver a message that originated in that time-frame, and all the messages that are in transit at the beginning of the time-frame are delivered before the end of the time-frame.

Let $ph.j.j$ at some state $S$ denote the number of distinguished events of $j$ that happened before $S$. Let $ph.j.j = x$ at state $S$. From the communication pattern above, for any process $k$, $ph.k.j$ is in the range $\{x - M \ \ldots \ x\}$ in $S$. For any message $m$ that is in transit in $S$, $ph.m.j$ is in the range $\{x - 2M \ \ldots \ x\}$.

**Definition.** $P_D$ satisfies $comm(M, l)$ iff (1) every computation of $P_D$ satisfies the communication pattern stated above, and (2) between any two adjacent distinguished events, the number of `send`, `receive` or `local-event` calls with *flag = True* is less than $l$, $l \in Z^+$. □

*Observation.* If a program satisfies $comm(M, l)$, then it also satisfies $comm(M', l')$ for any $M' \geq M$ and $l' \geq l$. □

As is the case with the comparison predicate, a large class of applications satisfy $comm(M, l)$ for some $M$ and $l$. For example, any phase-based application where each phase uses a standard communication pattern such as a diffusing computation over FIFO channels or heartbeat messages over time bounded channels satisfies *comm*. Also, if the application is not phase-based, the application designer can still identify (by studying the application or via experimentation) suitable values for $M$ and $l$ by considering bounds on process speed and message delay.

---

**Case study (continued).** To illustrate the use of the communication pattern, we show that $\text{RA}_D$ satisfies the comparison predicate with $M = 2$ and $l = 2$.

**Lemma 2.** $\mathrm{RA}_D$ satisfies $comm(2, 2)$.

**Proof.** $\mathrm{RA}_D$ satisfies $comm(M, l)$ with $l = 2$, since for any process $j$ there is only one event, the request of $j$, that is freshly timestamped between two adjacent distinguished events of $j$. In order to prove that $\mathrm{RA}_D$ satisfies $comm(M, l)$ with $M = 2$ we show below that within a time-frame of any two consequitive distinguished events of $j$:

1. all the processes deliver a message from $j$ that originated in that time-frame,
2. all the messages that are in transit at the beginning of the time-frame are delivered before the end of the time-frame.

(1) Process $j$ performs its $x^{th}$ distinguished event after it completes the critical section for $x^{th}$ time. Before $j$ performs its $(x+1)^{th}$ distinguished event to complete critical section for the $(x+1)^{th}$ time, $j$ sends a request message $m$ and receives a reply for the same. Therefore, for any process $k$, there exists an event $e_k$ (created by the reception of $m$) such that the $x^{th}$ distinguished event at $j$ happened before $e_k$, and $e_k$ happened before the $(x+1)^{th}$ distinguished event of $j$.

(2) Let $m'$ be a message in transit from $i$ to $k$ when $j$ performs its $x^{th}$ distinguished event. If $m'$ is a request message, $i$ is requesting for critical section and if $m'$ is a reply message then $k$ is requesting for critical section. Unless $m'$ is delivered, the requesting process cannot access its critical section. Using the result of bounded overtaking by Ricart-Agrawala, we now show that before $j$'s $(x+2)^{th}$ distinguished event, $m'$ is delivered. For simplicity, let us assume that $m'$ is a request message. If $m'$ is a reply message, the same argument applies with $i$ replaced by $k$. Ricart-Agrawala have shown that if $i$ is requesting for critical section then $j$ cannot access its critical section twice before $i$ accesses its critical section. When $j$ requests the critical section $(x+1)^{th}$ time, $i$ is already requesting for critical section. Therefore, before $j$ can enter the critical section for the $(x+2)^{th}$ time, $i$ must access its critical section and, hence, $m'$ must be delivered. In sum, any message in transit when $j$ performs its $x^{th}$ distinguished event, is delivered before $j$'s $(x+2)^{th}$ distinguished event.

It follows from (1) and (2) that $\mathrm{RA}_D$ satisfies the communication pattern $comm(M, l)$ with $M = 2$. □

## 4.2 Bounded-Space RVC Implementation

The implementation of RVC extends that of VC as follows. Each $j$ maintains a vector of phases, $ph.j$, in addition to its vector clock $vc.j$ . The $k^{th}$ entry in $ph.j$, $ph.j.k$, represents the latest information $j$ has about $ph.k.k$ . The contract $R(m,n)$ and $comm(M,l)$ is an argument of the `constructor` of RVC, which computes the *phase_bound* to be $max(m+n-1\ ,\ 3*M+1)$ and sets and the *clock_bound* to be $l$ (for justification of these values see the proof of Theorem 3).

The `send` and `local-event` operations implementation are similar to those of VC, except that the `increment-clock` is modified to use a bounded domain of values and roll over to start from 0 when the bound is reached. When process $j$ performs the `receive` operation to receive message $m_l$, it compares $ph.j.k$ with $ph.m_l.k$ to determine whether message $m_l$ carries more recent information regarding the phase about $k$. From $comm(M,l)$, $ph.m_l.k$ can carry information about at most $M$ new resets of $k$. Since $ph.j.k$ is bounded, we consider two cases (1) $ph.j.k < ph.m_l.k$ and (2) $ph.j.k > ph.m_l.k$ to determine if $m_l$ is carrying newer information regarding the phase about $k$. In case (1), $ph.m_l.k$ must be less than $ph.j.k+M+1$. In case (2), where the phase about $k$ has rolled over, $ph.m_l.k+phase\_bound$ must be less than $ph.j.k+M+1$. In either of the two cases, $m_l$ is carrying more recent information about $k$ and, hence, $j$ copies $ph.m_l.k$ and $vc.m_l.k$. If $ph.m_l.k = ph.j.k$ then $j$ simply updates $vc.j.k$ as it would update in VC. Finally, if $ph.m_l.k$ carries older information than $ph.j.k$, $j$ leaves $ph.j.k$ and $vc.j.k$ unchanged. Just as the `receive` operation uses $comm(M,l)$, `happened-before` operation uses $R(m,n)$ to decide the causality relation between events. The comparison predicate $R(m,n)$ determines how far $ph.e_j.j$ and $ph.f_k.j$ can be if $e_j$ and $f_k$ are compared. Similar to the `receive` operation, `happened-before` considers three cases (1) $ph.e_j.j = ph.f_k.j$, (2) $ph.e_j.j < ph.f_k.j$ and (3) $ph.e_j.j > ph.f_k.j$ to determine which event is aware of more resets of $j$. `Happened-before`$(e_j,\ f_k)$ returns true if $f_k$ is aware of more resets of $j$ than $e_j$, and returns false if $e_j$ is aware of more resets of $j$ than $f_k$. For the case where $e_j$ and $f_k$ are aware of equal number of resets of $j$, the `happened-before` operation uses $vc$ to decide whether or not $e_j$ $\underline{hb}$ $f_k$. Finally, the `nonblocking-reset`$(j)$ sets $vc.j.j$ to 0 and sets the $ph.j.j$ to $(ph.j.j+1)$ *mod phase_bound*.

**Bounded-Space RVC**

`constructor` ( $p\_id\_list$, $R(m,n)$, $M$, $l$ )
($\forall j,k : j,k \in p\_id\_list : vc.j.k := 0, ph.j.k := 0$);
$phase\_bound := max(m+n-1\ ,\ 3*M+1)$;
$clock\_bound := l$

`send` ( $j$, $e_j$, $m_j$, *flag* )
if (*flag*) then increment-clock ( $j$ );
$ph.m_j, vc.m_j := ph.j, vc.j$;
$ph.e_j, vc.e_j := ph.j, vc.j$

`receive` ( $j$, $e_j$, $m_l$, *flag* )
($\forall k : k \neq j :$
  if ( $(ph.j.k < ph.m_l.k \ \wedge \ ph.j.k + M + 1 > ph.m_l.k)$
    $\vee (ph.j.k > ph.m_l.k \ \wedge$
      $ph.j.k \geq ph.m_l.k + (phase\_bound - M))$ )
  then $ph.j.k, vc.j.k := ph.m_l.k, vc.m_l.k$
  elseif $(ph.j.k = ph.m_l.k)$
      then $vc.j.k := max(vc.j.k, vc.m_l.k)$
      else /* don't update $ph.j.k$ and $vc.j.k$ */ );
if (*flag*) then increment-clock ( $j$ );
$ph.e_j, vc.e_j := ph.j, vc.j$

`local-event` ( $j$, $e_j$, *flag* )
if (*flag*) then increment-clock ( $j$ );
$ph.e_j, vc.e_j := ph.j, vc.j$

`happened-before` ( $e_j$, $f_k$ ) : *boolean*
return(
    $(ph.e_j.j = ph.f_k.j \ \wedge \ vc.e_j.j \leq vc.f_k.j)$
  $\vee$ $(ph.e_j.j < ph.f_k.j \ \wedge \ ph.e_j.j + n > ph.f_k.j)$
  $\vee$ $(ph.e_j.j > ph.f_k.j \ \wedge \ ph.e_j.j \geq ph.f_k.j + m)$ )

`increment-clock` ( $j$ )
$vc.j.j := (vc.j.j + 1)\ mod\ clock\_bound$

`nonblocking-reset` ( $j$ )
$ph.j.j, vc.j.j := (ph.j.j + 1)\ mod\ phase\_bound,\ 0$

**Theorem 3.** *(substitutability of VC with bounded-space RVC)* Let $P$ be a well-formed client of VC and $P_D$ satisfy the comparison predicate $R(m,n)$ for some $m, n \in Z^+$, and the communication pattern $comm(M,l)$ for some $M, l \in Z^+$.
Then a bounded-space RVC with the constructor ( $p\_ids$, $R(\ m,\ n)$, $M$, $l$ ) is substitutable for $VC$ in $P$.

**Proof.** This proof consists of two parts. In the first part, we show that the condition $phase\_bound \geq max(m+n-1\ ,\ 3M+1)$ is sufficient for RVC to update the clock values correctly when $P_D$ uses RVC. Then, in the second part, we show that that condition is also sufficient for RVC to keep track of the causality relation correctly when it is used by $P_D$. More specifically, in the second part, we show that if the contract is satisfied then any two events $e_j$ and $f_k$ that $P$ compares, "happened-before($e_j$, $f_k$)" of RVC returns the same result as "happened-before($e_j$, $f_k$)" of VC. From the definition of substitutability in Section 2, it follows that RVC *is substitutable for* VC *in* $P$.

1) Note that when using RVC, $P_D$ executes a nonblocking reset event for its distinguished events and $vc.j.j$ is set to 0. From $comm(M, l)$, the value of $vc.j.j$ can never exceed $l$. The value of $vc.j.k$ is set only from another $vc$ value and, hence, there is no overflow on variable $vc$. The only thing that remains to show is that when a process $k$ receives message $m$ such that $ph.k.j$ and $ph.m.j$ are different, $k$ still updates the clock values correctly.

As in Lemma 1, we introduce an unbounded auxiliary variable, $ph'$ such that $ph'.k.j$ denotes the the number of distinguished events of $j$ that happened before the current event at $k$. Let $ph'.k.j = x$ when $k$ receives a message $m$. By our communication pattern assumption, $ph'.j.j$ is at most $x+M$ and, hence, $ph'.m.j$ is at most $x+M$.

Also, when $k$ receives message $m$, $ph'.j.j$ is at least $x$. When $j$ executes a distinguished action for the $x^{th}$ time, by our communication pattern, we have $\forall k :: ph'.k.j$ can be in the range $\{x - M,\ \ldots\ , x\}$, and $ph'.m.j$ for any message $m$ that is in transit can be in the range $\{x - 2M,\ \ldots\ , x\}$. It follows that $ph'.m.j$ is at least $x - 2M$.

Combining the above results, we have: if $ph'.k.j$ is $x$ when it receives message $m$ then $ph'.m.j$ is in the range $\{x - 2M,\ \ldots\ , x + M\}$. Upon reception of $m$, $k$ should change $ph'.k.j$ to $ph'.m.j$ provided $ph'.m.j$ is in the range $\{x,\ \ldots\ , x+M\}$. Otherwise, it should leave $ph'.k.j$ unchanged. From the receive action, if $k$ can distinguish between the values $\{x - 2M,\ \ldots\ , x + M\}$, $k$ updates $ph.k.j$ exactly in this way. In other words, if the domain of $ph.k.j$ is at least $3M + 1$, the clocks are updated correctly.

2) Let $e_j$ and $f_k$ be any two events that $P$ compares. ( $PR^m.e_j\ \underline{hb}\ f_k \ \wedge\ \neg(NX^n.e_j\ \underline{hb}\ f_k)$ ) holds since $P$ satisfies $R(m, n)$. We again

introduce the unbounded auxiliary variable "$ph'$" as in case 1. It follows from the comparison predicate $R(m, n)$ that $ph'.f_k.j$ is in the range $ph'.e_j.j$–($m$–1) ... $ph'.e_j.j$+($n$–1). Moreover, from the definition of $\underline{hb}$ and $ph'$, we have $(e_j \ \underline{hb} \ f_k) \equiv ph'.e_j.j < ph'.f_k.j \ \vee \ (ph'.e_j.j = ph'.f_k.j \wedge vc.e_j.j \leq vc.f_k.j)$

Note that the bounded phase variable $ph.j.k$ of RVC is equal to $ph'.j.k \ mod \ phase\_bound$, where $phase\_bound \geq (m+n-1)$ since $phase\_bound \geq max(m+n-1 \ , \ 3M+1)$. Recall that $e_j$ happened before $f_k$ iff either (1) $ph'.e_j.j = ph'.f_k.j$ and $vc.e_j.j \leq vc.f_k.j$ or (2) $ph'.e_j.j < ph'.f_k.j$. Using the bounded phase, these conditions are captured in the "happened-before" operation of RVC as follows:

- Since $ph'.f_k.j$ is in the range $ph'.e_j.j-(m-1)$ ... $ph'.e_j.j + n-1$ and $ph.j.k = ph'.j.k \ mod \ phase\_bound$, $ph'.e_j.j = ph'.f_k.j$ iff $ph.e_j.j = ph.f_k.j$. Thus, the first disjunction of the "happened-before" operation of RVC captures (1).
- If $ph'.e_j.j \leq ph'.f_k.j$, then there are two possibilities: $ph.e_j.j < ph.f_k.j$ or $ph.e_j.j > ph.f_k.j$:
  - If $ph.e_j.j < ph.f_k.j$ then $ph'.e_j.j < ph'.f_k.j$ iff $ph.f_k.j \leq ph.e_j.j+(n-1)$. This condition is captured by the second disjunction of the "happened-before" operation of RVC.
  - If $ph.e_j.j > ph.f_k.j$ then $ph'.e_j.j < ph'.f_k.j$ iff $\neg(ph.f_k.j \geq ph.e_j.j - (m-1))$. This condition is captured by the third disjunction of the "happened-before" operation of RVC. □

---

**Case study (continued).**

**Theorem 4.** A bounded-space RVC with the constructor $(p\_ids, R(3,2), 2, 2)$ is substitutable for VC in RA.

**Proof.** This proof follows trivially from Lemmas 1, 2 and Theorem 3. □

Note that, the minimum $phase\_bound$ of RVC substitutable for VC in RA is $max(3+2-1 \ , \ 3*2+1) = 7$. Therefore, the resulting RA application uses only bounded space.

---

# 5 Stabilizing Resettable Vector Clocks

The stabilization of RVC consists of a local detection of the violation of the invariant for RVC followed by an invocation of the global reset operation of RVC. Even though the contract might be temporarily violated in the presence of faults, the stabilization of RVC assumes that the client will eventually start re-satisfying the contract. The implementation of the local detector exploits the communication pattern part of the contract. Recall that if the communication pattern holds then for every message $m$, $ph.m.j$ will be in the range $\{ph.j.j - 2M \ \ldots \ ph.j.j\}$ for any $j$. RVC locally detects the out of range messages and calls the "global-reset" operation whenever a process receives an out of range message.

**Local detection.** In the absence of faults, the following predicate is invariantly true in any computation of $P_D$.

$$\begin{array}{l}(\forall j, k : j \neq k : \ \ rc.m.k \in [rc.j.k \ominus 2M \ \ldots \ rc.j.k \oplus M] \\ \qquad\qquad\qquad\qquad \wedge\, rc.m.j \in [rc.j.j \ominus 2M \ \ldots \ rc.j.j])\end{array}$$

Above $\ominus$ and $\oplus$ are subtraction and addition operations under modulo *phase_bound* arithmetic. The predicate states that due to the communication pattern when any process $j$ receives a message $m$, the timestamp of $m$ is within some range of the clock values at $j$. Whenever this invariant is violated, i.e., whenever a process receives an "out of range message" with respect to its clock, the local detection calls the `global-reset` operation to correct the component state. We insert this detector at the beginning of the `receive` operation.

**Bound on the number of phases.** If the "*phase_bound*" is not sufficiently large, there may exist a cycle among the processes which causes the processes to bump up their phase values infinitely without the local detection mechanism firing. For stabilizing fault-tolerance we increase the *phase_bound* for RVC to be at least "$max(m + n - 1, (BE + 2N - 1)M + 1)$" where $B$ is the bound on the channel capacity, $E$ is the number of channels, $N$ is the number of processes, and $M$, $m$, and $n$ are contract parameters as before.

Note that rather than using a local detection, one could alternatively perform global snapshots [7] to detect a violation of the invariant for RVC. Although it would improve the bound on the number of phases, we rule out this solution to avoid the communication overhead it introduces.

**Theorem 5.** The modified bounded RVC is stabilizing fault-tolerant.

**Proof sketch.** The proof for stabilization of the modified RVC consists of two parts. In the first part, we show that since the modified *phase_bound* is sufficiently large, there cannot be any cycle among the *ph* variables for any $j$ and $k$. In the second part, we show that when the component is in a faulty state, either the detector fires and the component is corrected with a global reset, or the component stabilizes at most within *phase_bound* nonblocking resets of every process. While the detailed proof is included in the appendix, we would like to note that the stabilization of RVC assumes that the application performs enough number of resets. Therefore we identify this condition as an additional contract to be satisfied by the application in order for RVC to recover from faults. □

**Theorem 6.** *(substitutability of VC with stabilizing and bounded-space RVC)* Let $P$ be a well-formed client of VC and $P_D$ satisfy the comparison predicate $R(m, n)$ for some $m, n \in Z^+$, and the communication pattern $comm(M, l)$ for some $M, l \in Z^+$.
Then a stabilizing fault-tolerant and bounded-space RVC with the constructor $(p\_ids, R(m, n), M, l)$ is substitutable for $VC$ in $P$.

**Proof.** This proof follows from Theorems 3 and 5. □

---

**Case study (continued).**

**Corollary 7.** A stabilizing fault-tolerant and bounded-space RVC with the constructor $(p\_ids, R(3, 2), 2, 2)$ is substitutable for VC in RA. □

---

# 6 Case Study (Continued): Stabilizing RA

In this section, we use the method described in the Introduction to design stabilizing tolerance to the RA application which consists of VC and its client (main module in RA). As discussed in the Introduction, we consider three levels of faults (1) in the main module of RA, (2) in the VC component and (3) in the interface between RA and VC.

For level (1), the component needs to recover itself to a consistent state. For this purpose, the component provides one or more operations that restore itself to a consistent state. Thus, in level (1), the component invokes one of its operations to restore itself to a consistent state. Since the new state of the component may not be consistent with the state of the application, the component raises an exception to the application; the application can provide one or more exception handlers to deal with exceptions raised by the component.

For level (2), the application needs to restore itself to a consistent state. Moreover, it also needs to ensure that the state of the component is consistent with its own state. Since the application is not aware of the internal details of the component, the application can only invoke the operations of the component for this purpose. Therefore, for level (2), the application chooses to correct its own state and/or to force the component to change its state appropriately by calling operations in the component.

For level (3), which may arise even when the application and the component are both internally consistent (but they are mutually inconsistent), detection is performed by the component or the application. Here, the method optimistically tries to deal with the fault by assuming that the application and the component are in internally consistent states. Therefore, if the component detects an interface fault, it raises an exception to the application. The application, upon receiving such exception or upon detecting an interface fault, invokes component operation(s) so that the component reaches another consistent state whereby mutual consistency is regained. (Should the optimistic assumption of internal consistency be invalid, the mechanisms that deal with levels (1) and (2) will eventually resolve the internal inconsistency.)

We discuss these three levels in Section 6.1, 6.2 and 6.3 respectively.

## 6.1 Stabilization of the main module

In order to design stabilizing-tolerance to the main module in RA, we exploit its specification. RA ensures that once a process requests for CS, it is guaranteed to enter CS within a certain period of time (i.e., (N-1) * (the maximum time it takes to grant CS to any process + the maximum time any process can stay in CS) ). Therefore, we can detect any inconsistency in the main module by using a "timeout" mechanism. Whenever a process $j$ calls the *REQUEST CS* action, we start the timer for $j$, and whenever $j$ is *granted the CS* we reset this timer. That is, we append the line **start-timer(j)** to *Request CS* action, and append **reset-timer(j)** to *Enter CS* action of RA.

When a timeout occurs for some process, a recovery is possible by using a global reset that resets the main module (the main module in turn resets the RVC) to a predetermined consistent state. However, we can achieve a lower-cost solution as follows. The detection of a timeout implies that a process is incorrectly deferred by some other process or a request or reply message is lost. After executing a *Release CS*, a process $j$ cannot incorrectly defer any process since it does not defer any process at all. Note that if we execute a *Release CS* for any process that timeouts, after some time there will not be any process $j$ incorrectly deferred by another process $k$; either $j$ will timeout and perform a new *Request CS*, or $k$ will timeout (or execute CS) and undefer $j$. Therefore, we add the following action to RA.

$$\boxed{\{\ \textit{timeout(j)}\ \} \longrightarrow \quad \textit{Release CS}(j);\ \textit{Request CS}(j)}$$

## 6.2 Stabilization of VC

By using Corollary 7 we substitute VC with a bounded-space and stabilizing-tolerant RVC in RA. Note that the stabilization of RVC relies on the assumption that its client will perform a sufficient number of reset events at every process, in turn it guarantees that within that many reset events it starts capturing the causality relation correctly. Since the main module never ceases to perform reset events for any process (even if

a process is incorrectly deferred from entering CS, a timeout leads to a *Release CS* and therefore a reset event), the RVC component stabilizes within some number (i.e., *phase_bound*) of reset events at every process. Also if RVC detects an inconsistency, it invokes its `global-reset` operation and raises an exception to the main module which in turn releases CS in order to start fresh and synchronized with RVC.

## 6.3 Stabilization of the interface invariant

The specification for RA includes the following condition:

$$(\forall j, k : j \neq k : (j\text{'s request}) \ \underline{hb} \ (k\text{'s request}) \Longrightarrow \text{happened-before}(REQ\_ts.j \ , \ REQ\_ts.k)\text{=True} \ )$$

Note that this condition is also an interface invariant between the main module and RVC. It may be the case that both the main module and RVC are internally consistent but the above condition is false; the requests of processes (i.e., $REQ\_ts$ variables) of the main module may have incorrect values rather than the values assigned to them by RVC.

Since we are only interested in stabilizing-tolerance (rather than a masking-tolerance where safety is never violated), we do not try to detect this inconsistency. Instead we ensure that in RA the inconsistencies at the interface are corrected eventually.

It may be the case that an interface fault is detected by RVC; the incorrect $REQ\_ts$ values invoke the local detection mechanism since they are out of the valid range. In this case, the application (hence the interface invariant) stabilizes as discussed in Section 6.2. If that is not the case, the interface invariant stabilizes after all the processes request for new CS executions, since the new requests will get a correct value from RVC.

In the above discussion, we assumed without loss of generality that both the main module and RVC has been stabilized, since otherwise Sections 6.1 and 6.2 would apply.

**Theorem 8.** RA with the modifications suggested in Sections 6.1, 6.2, and 6.3 is stabilizing fault-tolerant. □

# 7 Application: Stabilizing Online Predicate Detector

In this section, we demonstrate our method for designing fault-tolerance for VC applications on a conjunctive predicate detection program. First, in Section 7.1, we discuss the weak conjunctive predicate detection problem. In Section 7.2, we present Garg and Chase's weak conjunctive predicate detector program [12]. Then, in Section 7.3, we show how to bound the implementation of a weak conjunctive predicate detector by substituting VC with bounded RVC in the detector and by designing a wrapper around the predicate detector. Finally, in Section 7.4, we show how to make the detector itself stabilizing.

## 7.1 Global Predicate Detection

A global predicate of a program is a predicate on the state of all processes. A global predicate is either stable or transient in the program: it is stable iff it never turns false once it becomes true. Both the stable predicate detection [7] and unstable predicate detection problems [12, 13] have been considered in detail in the literature.

In this section, we focus on the detection of weak conjunctive predicates [13], i.e., transient predicates that are of the form $c1 \wedge c2 \wedge ...cn$, where $ci$ $(1 < i < n)$ is a local predicate on process $i$. The detection of weak conjunctive predicates is known to be sufficient to detect any global predicate on a finite state program, as well as any global predicate that can be written as a boolean expression of local predicates. Our detector differs from that in [12, 13] in that our detector is online, stabilizing-tolerant and has bounded implementation.

For a given computation and $\underline{hb}$ relation on that computation, we define:

- A *global cut* is a set of local events such that exactly one event is included from each process.
- Two global cuts are *overlapping* iff their intersection is nonempty, i.e., there is at least one common event between them.

- A global cut is *consistent* iff all the events in the cut are concurrent with each other.
- A *conjunctive predicate* is a predicate obtained by conjunction of the local predicates of all processes.

**The weak conjunctive predicate detection problem.** Given a computation and $\underline{hb}$ relation on that computation, for each consistent cut $C$ where the conjunctive predicate $Z$ is true, find a consistent cut $G$ such that $G$ overlaps with $C$ and $Z$ is true on $G$.

## 7.2 A Weak Conjunctive Predicate Detector

In this section, we present a weak conjunctive detector due to Garg and Chase [12] which works as follows. In their program, each client process maintains an event queue that consists of vector clock timestamps of events where the local predicate of that process was true. The detector circulates a token among processes and operates on the event queues. The token identifies a global snapshot where the conjunctive predicate is true. However, this snapshot may be inconsistent. Such inconsistencies are marked with a color. Hence, the token consists of two arrays $G$ and *color*, where $G$ denotes a (possibly inconsistent) cut of the system, and *color* is used to mark the inconsistencies (if any) in the cut. For any process $j$, $G.j$ denotes the minimum clock value of $vc.j.j$ for which a consistent cut, where the conjunctive predicate is true, may be found. Therefore, whenever process $j$ receives a token, if $G.j$ is greater than the value of $vc.j.j$ in the event at the head of the event queue, $j$ dequeues events from its event queue until $vc.j.j > G.j$ is true. After dequeuing events from the queue, $j$ updates $G.k$ for each process $k$ to be the maximum of $G.k$ and $vc.j.k$. Thus, this action modifies the value of $G.k$ to ensure that clock of $k$ will be concurrent with the event that $j$ just deqed from its event queue. If $j$ modifies the value of $G.k$ then it marks $color.k$ as red indicating that $k$ must advance its clock before a consistent snapshot can be obtained. Observe that by advancing in this manner, the value of $G.k$ denotes the minimum value of $vc.k.k$ for which a consistent cut may be found where the conjunctive predicate is true. When the cut $G$ is consistent (i.e., no red color), $j$ declares that the conjunctive predicate is detected.

In the implementation of the detector, we write $\hat{G}_i$ to denote an auxiliary event at process $i$ whose timestamp is $G$. Also, since we are interested in detecting the predicate for only one of the overlapping cuts, after the predicate is detected for a cut $C$, we remove all of the events in $C$ from their corresponding event queues. Thus, the predicate detector is shown in Figure 2.

**Weak conjunctive predicate detector**

{Reception of the token by a process $j$} $\longrightarrow$
  while $(\neg(\hat{G}_j \ \underline{hb} \ (ptr.j)_j))$
    $ptr.j$ := GetNextEvent($event\_queue.j$);
  $G.j, color.j := rc.(ptr.j).j, green$ ;
  if $((\forall i : i \neq j : \neg(\hat{G}_i \ \underline{hb} \ (ptr.j)_i) \wedge color.i = green))$
  then SignalDetection();
  else $(\forall i : i \neq j :$ if $(\hat{G}_i \ \underline{hb} \ (ptr.j)_i)$
    then $G.i, color.i := rc.(ptr.j).i, red$ );
  PropagateToken($j$);

{SignalDetection} $\longrightarrow$ //one instance found
  Record $G$;
  $(\forall j : ptr.j :=$ GetNextEvent($event\_queue.j$));

Figure 2: A weak conjunctive predicate detector implementation

## 7.3 Bounded Implementation of a Weak Conjunctive Predicate Detector

In order to bound the implementation of the conjunctive predicate detector of Garg and Chase, we need a bounded implementation of RVC. For this application we assume that 1) the client satisfies $comm(M, l)$ with $M = 1$ and for some $l$ and 2) between any two consecutive resets, $x$ and $x+1$, of any process $j$, $j$ will deliver a message $m_k$ from every other process $k$. This communication pattern can be generalized, however, for brevity, we omit that generalization.

To ensure that the events compared by the conjunctive predicate detector satisfy a comparison relation $R(m, n)$ for some $m$ and $n$, we introduce the following synchronization events to the event queues:

- Whenever the local event at a process $j$ is a reset event, $j$ places, in its event queue, the timestamp of the event which is just before this reset event. This event is tagged as a "synchronization event" rather than an event where the local predicate is true.

- Whenever process $j$ learns about the reset event of another process $k$ for the first time, $j$ inserts in its event queue the timestamp of the receive event and tags that event as a synchronization event. Note that if $j$ learns about multiple resets of $k$ within the same message, then for each of these resets $j$ places separate timestamps in its queue.

**Lemma 9.** The weak conjunctive predicate detector satisfies $R(3, 2)$.

**Corollary 10.** From Theorem 3, it follows that VC is substitutable with a bounded RVC with a phase of 4 (i.e., $max\ (3+2-1\ ,\ 3*1+1)$) in the weak conjunctive predicate detector. □

After adding these synchronization events to the event queues, we should also design a wrapper around the predicate detector to take care of these synchronization events. In our implementation of the wrapper for the detector, in addition to $G$ and *color* variables maintained in the token, we append another variable $X$ to the token in order to differentiate between the synchronization events and the events where the local predicate is true. $X.j$ is false iff the event at the head of the event queue of a process $j$ is a synchronization event. The conjunctive predicate is detected when $G$ is a consistent cut and $(\forall i : X.i = true)$. Since we are interested in detecting the predicate for only one of the overlapping cuts, after the predicate is detected for a cut $C$, we mark all of the events in $C$ as invalid by assigning $X.i$ to $false$ for all $i$. Whenever the token is sent to the next process on the ring, we piggyback the variable $X$ to the token.

If $C$ is a consistent cut such that the event of process $j$ in $C$, say $e_j$, is a synchronization event then in any subsequent consistent cut where the conjunctive predicate is true, $j$ must dequeue $e_j$. By way of contrast, if $e_j$ is not a synchronization event then there may exist a consistent cut where the conjunctive predicate is true and $e_j$ is included in that cut. It follows that, if $e_j$ is not a synchronization event then $e_j$ should not be dequeued

at this stage.

While designing the wrapper, we use only the *GetNextEvent(event_queue.j)* operation of the predicate detector which sets the head of the queue ($ptr.j$) to the next event of the event queue. In the implementation, we write **preddetector::X** to denote an operation X on the predicate detector program presented in Section 7.2. Thus, our bounded-space predicate detector program is as follows.

**Bounded-space weak conjunctive pred. detector**

{Reception of the token by a process $j$} $\longrightarrow$
  if $(\neg X.j)$ then
      $ptr.j :=$ GetNextEvent($event\_queue.j$);
  `preddetector::{Reception of the token by j};`

{PropogateToken($j$)} $\longrightarrow$
  if ($ptr.j$ is an event where the local predicate is true)
  then $X.j := true$
  else $X.j := false$;
  `preddetector::{PropogateToken(j)};`

{SignalDetection} $\longrightarrow$
  if $((\forall i :: X.i))$ then
      `preddetector::{SignalDetection};`

To get a bounded detector, the detector should be able to block the client program so that the event queues at every process remain bounded. In particular, we restrict the client program so that for any process $j$ the number of reset events of $j$ in the event queue at $j$ is bounded by some integer $\Delta$, $\Delta \geq 1$.

## 7.4 Stabilizing Online Predicate Detector

In this section, we use the method described in the Introduction to design stabilizing-tolerance to the online predicate detector application which consists of VC and its client (main module in predicate detector).

**Stabilization of the main module.**

Since the event queues at every process remain bounded so that for any process $j$ the number of reset events of $j$ in the event queue at $j$ is at most $\Delta$, from the comparison predicate that the predicate detector satisfies ($R(3,2)$) we increase the domain of $ph.j.k$ to have at least $\Delta+4$ distinct values (from 0 to $\Delta+3$). With this modification, we prevent the following scenario: Let the head of the event queue at process $j$ has the phase value $x$, and let there exist another event in the middle of the queue with phase value $x$. When a transient fault deletes the upper half of $j$'s event queue, the predicate detector might not have a way to detect this since the event queue at $j$ may incorrectly synchronize with the other event queues.

We also require each process to check the following conditions. Whenever process $j$ receives the token, it checks whether its event queue satisfies the following four consistency requirements:

1. For any two consecutive events, $e$ and $f$, $ph.f.k$ is either equal to $ph.e.k$ or $ph.e.k$+1.
2. There are at most $\Delta$ reset events of process $j$ in the queue.
3. There are at most $\Delta+2$ different phase values for process $k$, $k \neq j$.
4. For all $k$, the phase value of $k$ in the event at the head of the queue - Hide quoted text - at $j$ is in the range $G.k-2...G.k+1$. It does the same check whenever it considers a new event from the queue.

Note that in the absence of faults, all the conditions described above will be satisfied. Therefore, if either of these conditions is not satisfied, $j$ can conclude that the internal state of the detector is perturbed by a transient fault and invokes a global reset operation to reset the event queues and calls the `global-reset` operation for RVC.

**Stabilization of VC.** By using Theorem 6 we substitute VC with a bounded-space and stabilizing-tolerant RVC. Note that we assume the contract for the stabilizing-tolerant RVC is satisfied by the client program of the predicate detector. If RVC detects an inconsistency and invokes its `global-reset` operation self-reflectively, it notifies the main module about this decision which in turn resets the event queues for all processes in order to start fresh and synchronized with RVC.

**Stabilization of the interface invariant.** The interface invariant between the detector and stabilizing and bounded RVC is trivial; the detector relies on RVC to correctly keep track of the causality relationship eventually. Thus, after RVC stabilizes, the interface invariant also stabilizes.

**Theorem 11.** The bounded-space weak conjunctive predicate detector presented in Section 7.3 is stabilizing fault-tolerant. □

# 8 Concluding Remarks

Broadly speaking, this paper addresses issues in the systematic design of fault-tolerant component-based applications. Our experience is that by making the components themselves fault-tolerant and by exploiting a contract between the application and the component, design of application fault-tolerance is simplified. In other words, compositional design of fault-tolerance is potentially simpler than monolithic design of fault-tolerance.

More specifically, we presented a generalization of VC, namely RVC, based on the observation that in many applications causality comparisons are between events that occur within some number of adjacent "phases" of application execution. This observation has led us to introduce nonblocking resets in order to enable the reuse of timestamps.

We solved two problems, the first of which deals with how to design a bounded-space and fault-tolerant RVC. The second problem deals with how to design fault-tolerance in an application that uses VC, based on the substitution of VC with RVC.

For the first problem, we argued that applications need to satisfy a nontrivial contract in order to use a bounded-space RVC. We identified one such contract, which consists of a comparison predicate $R(m, n)$ and a communication pattern $comm(M, l)$. For this contract, we presented a bounded-space and stabilizing fault-tolerant RVC implementation. The space bound depends on the parameters $m, n, M, l$ and the system size (i.e., $B, E, N$); the stronger the contract, i.e. the smaller the parameter values, the smaller the space bound. The contract is readily satisfied by phase-based applications where events are compared only within some number of adjacent phases and each phase uses

a standard communication pattern such as a diffusing computation over FIFO channels or heartbeat messages over time-bounded channels. In these applications, non-blocking resets are inserted when a process moves from one phase to another. Once the designer determines where the reset points are, it is easy to verify for what parameter values the contract is satisfied.

Elsewhere [4], we have identified other contracts that an application may satisfy in order to use RVC. More specifically, we have shown that if the communication pattern requirement is strengthened so that between $M$ resets of $j$, each process directly receives a message from $j$, the size of each entry in the RVC can be made independent of the system size. Moreover, if the communication pattern is dropped (i.e., if $M = l = \infty$), it is still possible to bound the timestamps, although the implementation itself may be unbounded.

For the second problem, we argued that fault-tolerance in a component-based application involves dealing with faults in three different levels: in the component, in the application, or in the interface between the two. We presented a method that deals with these levels, and demonstrated the method by transforming Ricart-Agrawala's mutual exclusion program [19] to be both bounded-space and stabilizing fault-tolerant. We have also used the method to design an online, bounded-space and stabilizing fault-tolerant version of Garg-Chase's transient predicate detector [12]. In both cases, prior solutions had lacked bounded-space and stabilizing fault-tolerance.

Our work suggests several directions for future work. For clock components, the problems to be studied include: (i) the design and use of bounded-space logical clock component; (ii) the design of multitolerance [2] in the RVC component, e.g. in the presence of limited faults such as process crashes, masking fault-tolerance is provided —causality is thus correctly tracked despite these faults— but in the presence of more general faults, only stabilizing fault-tolerance is provided; (iii) the design of local correction instead of global correction for RVC stabilization; and (iv) the design of fault-containing RVC, where the effect of the faulty state is contained to only a few processes. Regarding the problem of designing fault-tolerant applications, future work includes (i) methods that exploit the client-component contract to prove substitutability of a component by a fault-tolerant version thereof, and (ii) scalable techniques for verification of fault-tolerance in large

scale applications.

# References


[1] Y. Afek and S. Dolev. Local stabilizer. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 287, 1997.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[3] A. Arora, S. S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 269–278, August 2000.

[4] A. Arora, S. S. Kulkarni, and M. Demirbas. Resettable vector clocks. Technical report, Ohio State University, January 2000.

[5] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded. *Infocom*, June 1994.

[6] K. Birman and T. Joseph. Reliable communication in the presence of failures. *"ACM Transactions on Computer Systems"*, 5(1):47–76, 1987.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[8] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[9] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[10] M. A. F. J. Torres-Rojas. Plausible clocks: Constant size logical clocks for distributed systems. *Proceedings of WDAG*, pages 71–88, 1996.

[11] J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

[12] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. *International Conference on Distributed Computing Systems*, pages 423–430, June 1995.

[13] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

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

[15] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[16] S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maintaining potential causality. In *Proceedings of Principles of Distributed Computing (PODC)*, pages 231–239, 1991.

[17] M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock based self-stabilizing distributed mutual exclusion. *16th International Conference on Distributed Computing Systems*, May 1996.

[18] A. Mostefaoui and O. Theel. Reduction of timestamp sizes for causal event ordering. Technical Report Reserch Report 1062, Irisa, 1996.

[19] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1991.

[20] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. *Proceedings of the International Workshop on Distributed Algorithms*, pages 219–232, 1989.

[21] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, 1992.

[22] L.-H. Yen and T.-L. Huang. Resetting vector clocks in distributed systems. *Journal of Parallel and Distributed Computing*, 43(1):15–20, 1997.

**Anish Arora.** Anish Arora is a Professor of Computer Science at the Ohio State University. His research is on fault tolerance, security, and timeliness properties of systems, especially distributed and networked systems of large scale. Recent case studies in his research have centred on sensor networking and home networking, with support from Darpa, NSF, and Microsoft Research. Arora is a leading expert in self-stabilization, and has chaired or co-chaired seminars and symposia in this area in 1998, 1999, 2000, and 2002. He is program co-chair of the 25th International Conference on Distributed Computer Systems. Arora received the B. Tech. degree from the Indian Institute of Technology at New Delhi and the Master's and Ph.D. degrees from the University of Texas at Austin, all in computer science. From 1989 to 1992, he worked at the Microelectronics and Computer Technology Corporation (MCC) in Austin, TX. (URL: http://www.cis.ohio-state.edu/ anish).

**Sandeep Kulkarni.** Kulkarni received his B.Tech from Indian Institute of Technology, Mumbai, India in 1993. He received his M.S. and Ph.D. degrees from The Ohio State University in 1994 and 1999 respectively. He is currently an assistant professor at the Computer Science Department of Michigan State University. His research interests include fault-tolerance, security, reliability, component-based design, and formal methods. He is a receipient of NSF CAREER award.

**Murat Demirbas.** Demirbas received his B.Tech from Middle East Technical University, Ankara, Turkey in 1997. He received his Master's and Ph.D. degrees from The Ohio State University in 2000 and 2004, respectively. He is currently a post-doc with the Theory of Computing Group at MIT. Murat's research interests are in the area of distributed systems, sensor networks, and fault tolerance. In 2002, he is awarded with an *outstanding researcher award* from the Computer Science Department of The Ohio State University, and with *a best paper award* from the International Conference on Distributed Computing Systems.

# Appendix

### Implementation of the Vector Clock Component.

In the vector clock component, each process $j$ maintains a vector clock $vc.j$ . The $k^{th}$ entry in $vc.j$ , $vc.j.k$ , represents the latest knowledge of process $j$ about $vc.k.k$ . The vector clock component implementation is as follows.

**Vector Clock Component**

`constructor ( p_id_list )`
$(\forall j, k : j, k \in p\_id\_list : vc.j.k := 0);$

`send ( j, e_j, m_j, flag )`
if (*flag*) then increment-clock ( $j$ );
$vc.e_j, vc.m_j := vc.j, vc.j$ ;

`receive ( j, e_j, m_l, flag )`
$(\forall k : vc.j.k := max(vc.j.k, vc.m_l.k)$ );
if (*flag*) then increment-clock ( $j$ );
$vc.e_j := vc.j$ ;

`local-event ( j, e_j, flag )`
if (*flag*) then increment-clock ( $j$ );
$vc.e_j := vc.j$ ;

`happened-before ( e_j, f_k )` : *boolean*
return $(vc.e_j.j \leq vc.f_k.j)$

`increment-clock ( j )`
$vc.j.j := vc.j.j + 1$

**Lemma 0.** The following condition is invariantly true in all computations that use the vector clock component.

$$\forall e_j, f_k :: (e_j \ \underline{hb} \ f_k) \ \equiv \ (vc.e_j.j \leq vc.f_k.j)$$

□

**Proof of Theorem 5.**

Observe that when process $k$ receives $m$, it invokes local detection if $ph.m.j$ is not in $\{ph.k.j-2M, ph.k.j+M\}$. If any process invokes local detection, the system is globally reset to a consistent state. Next, we show that if the number of distinct phases is large enough, either local detection will fire at some process or the system will be in a consistent state.

First, we identify how many different phase values about a process, say $j$, can exist simultaneously; For each process $k$, there can exist one value, namely $ph.k.j$, and for each message $m$ there can exist one value, namely, $ph.m.j$. Thus, there can be at most $BE + N$ values in the system.

Let $ph.j.j = x$ in the initial state. For the following discussion, consider what happens when $j$ goes from its $x^{th}$ reset to its $(x+M)^{th}$ reset. In this case, only $j$ can generate new phase values with respect to $j$ and they are in the range $\{x..x+M-1\}$. No other process can generate a new value for the phase about $j$. Also, assume that local detection does not fire at any process, and the phase is with respect to $j$.

From the communication pattern, before $j$ does $(x+M)^{th}$ reset, each process creates an event that causally depends on the $x^{th}$ reset of $j$. It follows that there exists a neighbor $k$ such that $k$ receives a message $m_j$ such that $ph.m_j.j \in \{x, \ \ldots \ , x + M - 1\}$.

Since the local detection does not fire at $k$, we have: $ph.m_j.j \in \{ph.k.j–2M, \ \ldots \ , ph.k.j+M\}$. From the receive action on page 16, after delivering $m_j$, we have $ph.k.j \in \{x, \ \ldots \ , x+3M\}$. There are additional $BE+N–3$ phase values (everything except $j$, $k$ and $m_j$) in the system. If $k$ receives a message with these phase values, $k$ can increase its phase value by at most $M$ at a time. In other words, the phase of $k$ can increase up to $x+(BE+N)M$.

Consider process $i$ that receives a message, say $m_k$, from $k$. Let process $k$ have received $y$ messages before sending a message to $i$. It follows that $ph.m_k.j$ is in the range $\{x, \ \ldots \ , x+3M+yM\}$. Using the same argument in case of reception of $m_j$, when $i$ receives $m_k$, $ph.i.j$ is in the range $\{x, \ \ldots \ , x + 3M + yM + 2M\}$. There are additional $BE+N-3-y-1$ phase values in the system (the -1 is due to process $i$), and each of them can increase the value of $ph.i.j$ by at most $M$. It follows that phase of $i$ can increase up to $x+(BE+N+1)M$.

By induction on the number of processes, when $j$ does its $x+M^{th}$ reset, the phase value of any process is in the range $\{x, \ \ldots \ , x+(BE+2N-2)M\}$. Note, however, that when $j$ does the $(x+M)^{th}$ reset, the phase values of messages in transit may still be arbitrary.

When $j$ does $(x+2M)^{th}$ reset, by the same argument, the phase value of any process are in the range $\{x+M, \ \ldots \ , x+(BE+2N-1)M\}$. Moreover, the messages that were in transit when $j$ does $(x+M)^{th}$ reset have been delivered. Thus, the phase values of messages in transit are in the range $\{x, \ \ldots \ , x+(BE+2N-1)M\}$.

As long as there is no process or message whose phase value with respect to $j$ is in the range $x+(BE+2N)M$, the phase value of any process cannot increase beyond $x+(BE+2N-1)M$. Therefore, it suffices that the domain of phase values should be $(BE+2N)M+1$. If the domain of phase values is at least $(BE+2N)M+1$, eventually when $ph.j.j$ reaches $(BE+2N-1)M$, the system is in a consistent state. □

**Proof of Theorem 8.**

The following predicate, I, is invariantly true in any computation of bounded RA.

$I_1 = (\forall j,k : j \neq k : (j \in deferred\_set.k) \Longrightarrow$
less-than($REQ\_ts.k$ , $REQ\_ts.j$))
$I_2 = (\forall j,k : j \neq k : REPLY.j.k =$
last $REQ\_ts.k$ that $j$ has received from $k$)
$I_3 = (\forall j : CS.j \Longrightarrow$
$(\forall k : k \neq j : REQ\_ts.j = RECVD.j.k))$
$I_4 = (\forall j : hungry.j \Longrightarrow REQ\_ts.j.j =$ RVC$.j.j)$
$I_5 = (\forall j,k : j \neq k :$ ($j$'s request) $\underline{hb}$ ($k$'s request) $\Longrightarrow$
happened-before($REQ\_ts.j$ , $REQ\_ts.k$)=True )
$I = I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5$

From the timeout action, observe that each process will invoke the non-blocking reset operation unbounded number of times. From the proof of Theorem 5, eventually, the clock component will stabilize and, therefore, the clock values used by the RA will be consistent. For the following discussion, we assume that the clock component is consistent.

Whenever process $j$ receives a request message from $k$, predicate $I_2$ is truthified. Whenever process $j$ enters critical section, predicate $I_3$ is truthified. If predicate $I_1$ is false, i.e., if process $j$ is improperly deferred by $k$, then eventually $j$ will timeout, release the CS and send a new request to $k$. Upon reception of this request, $k$ will update $deferred\_set.k$ correctly and truthify $I_1$. Moreover, these predicates are closed in other actions. Thus, they continue to be true.

The predicate $I_4$ is corrected when the processes request for their next CS. Also if the resettable clock component detects an inconsistency and invokes a global reset, it notifies RA about this decision which in turn throws away the old $REQ\_ts$'s and calls the clock component to get fresh timestamps for its requests.

The predicate $I_5$ is truthified when both $j$ and $k$ have obtained fresh timestamps for their requests after the clock component has become consistent. □

**Proof of Lemma 9.**

Observe that our detector guarantees the following property: Any event $f_k$ at process $k$ (see Figure 3) is compared with an event $e_j$ at process $j$ iff $e_j$ is concurrent with $f_k$ (i.e.,

$e_2, e_3$ and $e_4$), or $e_j$ is the first event that happened after $f_k$ (i.e., $e_5$), or $e_j$ happened before $f_k$ and $ph.e_j.j = ph.f_k.j$ or $ph.e_j.j = ph.f_k.j - 1$ (i.e., $e_1$).

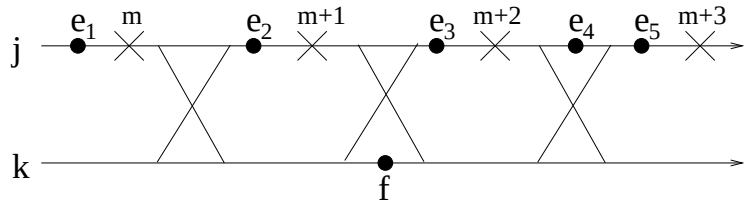


Figure 3: The comparisons required by the detector

Thus, for any two events $e_j$ and $f_k$ to be compared in any of the comparisons of our detector the following condition is satisfied :
$(ph.e_j.j \geq ph.f_k.j - 1 \ \vee \ ph.e_j.j \leq ph.f_k.j + 2) \quad \wedge \quad (ph.f_k.k \geq ph.e_j.k - 1 \ \vee \ ph.f_k.k \leq ph.e_j.k + 2)$

It follows, from the definition of $R$, that if the detector needs to compare any two events $e_j$ and $f_k$ then $R(3,2).e_j.f_k$ is true. □

**Proof of Theorem 11.**

The detector recovers from an arbitrary state in the following steps.

- Starting from an arbitrary state, as shown in [8], the token ring reaches a state where there is exactly one token.
- After one circulation of the token, for any process $j$, the value of $ptr.j$ is the same as the phase value of the event at the head of the queue at $j$. Moreover, when the token visits $j$, the values of $G.j, X.j, color.j$ are corrected according to the event at the head of the event queue at $j$.
- Without loss of generality, we can assume that the conditions described above are satisfied and that RVC is stabilized. Therefore, for any new events inserted in the event queues, the causality relation predicted by RVC is correct.
- If the phase value of $ptr.j.j$ is $x$ then the phase value in $ptr.k.j$ is either $x-2, x-1, x, or x+1$. Also, the phase values of $j$ in the event queue at $j$ are in the range $x..x + \Delta$. Therefore, the phase of values of $j$ in the event queue at $k$ are in the range $x-2..x + \Delta$. Since there are $\Delta+4$ values for the phase, all the values in the range $x-2..x + \Delta$ are distinct.
- Eventually, the events at the head of the event queues are consumed. Moreover, the causality relation among the new entries enqueued in the event queue is correctly captured by RVC. Therefore, eventually the detector reaches a state where all the events in the event queues satisfy the causality relation predicted by RVC.

- After the program reaches a state where all the above conditions are satisfied, the token circulation will eventually detect a consistent snapshot of the system (if one exists) where the conjunctive predicate is true. □

# Notation

| Symbols | |
|---|---|
| $i, j, k, l$ | processes |
| $e, f$ | events |
| $e_j$ | event at process $j$ |
| $m$ | message |
| $m_j$ | message sent by process $j$ |

| Variables | |
|---|---|
| $vc.j.k$ | vector clock at $j$ about $k$ |
| $ph.j.k$ | phase value at $j$ about $k$ |
| $rc.j.k$ | resettable vector clock at $j$ about $k$ |

| Functions | |
|---|---|
| $PR.e_j$ | PRevious reset (last reset at $j$ before $e_j$) |
| $NX.e_j$ | NeXt reset (first reset at $j$ after $e_j$) |

| Causal relations | |
|---|---|
| $e \underline{hb} f$ | $e$ happened before $f$ |
| $e \underline{co} f$ | $e$ and $f$ are concurrent |

| Propositional connectives (in decreasing order of precedence) | |
|---|---|
| $\neg$ | negation |
| $\wedge, \vee$ | conjunction, disjunction |
| $\Longrightarrow, \Longleftarrow$ | implication, consequence |
| $\equiv, \not\equiv$ | equivalance, inequivalence |

| First order quantifiers | |
|---|---|
| $\forall, \exists$ | universal, existential |