Resettable Vector Clocks [*]

Anish Arora [†]
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA
anish@cis.ohio-state.edu

Sandeep Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
sandeep@cse.msu.edu

Murat Demirbas
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA
demirbas@cis.ohio-state.edu

ABSTRACT

Vector clocks (VC) are an inherent component of a rich class of distributed applications. In this paper, we consider the problem of realistic —more specifically, bounded-space and fault-tolerant— implementation of these client applications. 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 demonstrate our method in the context of Ricart-Agrawala's mutual exclusion program.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems—*Distributed applications*; C.4 [**Computer Systems Organization**]: Performance of systems—*Fault tolerance, Reliability, availability, and serviceability*; D.2.13 [**Software**]: Software Engineering—*Reusable libraries*; D.2.4 [**Software**]: Software Engineering—*Programming by contract*

General Terms

Vector Clocks, Reset Events, Component Substitutability, Bounded-Space, Fault-Tolerance, 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 [7] and Mattern [11] independently proposed vector clocks that capture the causality relation between events. Vector clocks (VC) are extensively used in distributed applications, such as, distributed debugging [8], checkpointing and recovery, and causal communication [15].

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, which leads us to make two observations about VC. First, VC often consumes more space than is necessary by requiring that a "fresh timestamp" be associated with every send, receive and local event of the client application. But most clients causally compare only certain events, and VC correctness is preserved by freshly timestamping only these events; the remaining events could therefore reuse the current timestamp. Second, and more importantly, many applications are structured in phases and track causality only within a bounded number of adjacent phases. This suggests another way to reuse timestamps: augment VC with a "nonblocking reset" operation that lets an application process 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, often a bounded number of clock values suffice.

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, as Jayaram and Varghese [9] have shown, simultaneous occurrence of process crash and message reorder can drive applications into arbitrary states, including ones where all space is used up. Moreover, when simple faults such as reboot of the application process occur, the coordination between vector clocks may be disrupted and the tracking of the causality relation may become incorrect. 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 with-

[*]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 and a grant from Department of Computer Science and Engineering at Michigan State University.

[†]Arora is currently on sabbatical leave at Microsoft Research.

out 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; if an application does not satisfy the contract, the application should be easily modified so as to satisfy it. 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 stabilizing fault-tolerance.

The second problem deals with how to transform VC applications into fault-tolerant ones, using RVC 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, as follows.

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.)

Contributions of the paper. We give a bounded-space implementation of the VC component, and show how to make it stabilizing fault-tolerant. 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.

Finally, we use Ricart and Agrawala's mutual exclusion [14] as an illustration of our method for transforming a VC-based application to a bounded-space, stabilizing fault-tolerant version thereof. To the best of our knowledge, prior implementations for Ricart-Agrawala's mutual exclusion have lacked bounded-space and stabilizing tolerance.

Related work. Mostefaoui and Theel [13] 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. Also, their solution is neither bounded-space nor stabilizing fault-tolerant. By way of contrast, our solution achieves both.

Awerbuch, Patt-Shamir and Varghese [3] 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 [3] 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 [14] uses $O(log\ n)$ space per process, where n is the number of processes, and is not stabilizing fault-tolerant. Mizuno et. al. [12] gave a stabilizing version, but their solution uses unbounded state. Our solution uses $O(n\ \ log\ n)$ space per process.

Organization of this paper. In Section 2, we present the preliminaries. 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 stabilizing tolerant to transient faults in Section 5. In Section 6, we use the bounded-space and fault-tolerant RVC to provide a bounded and stabilizing fault-tolerant implementation of the mutual exclusion solution by Ricart and Agrawala [14]. Finally, we make concluding remarks in Section 7.

2. PRELIMINARIES

In this section, we discuss the system model, faults, stabilizing fault-tolerance, 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 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 [10] 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:

- for any two events e and f, if e and f are events on the same process and e occurred before f, then $e\ \underline{hb}\ f$
- for any two events e and f, if e is a send event in one process and f is the corresponding receive event in another process, 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.

Stabilizing fault-tolerance. A program P is *stabilizing fault-tolerant* 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 [6].

Finally, for any variable of the form $var.j$, j is the process where the variable is maintained. Depending upon the dimension of a variable, we use one or more indices. For example, if $var.j$ is an array 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` [1]

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.

[1] *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.

- `global-reset( )`: This operation performs a blocking reset for all processes and introduces extra communication messages. This operation is not required 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 $flag = True$. □

To illustrate the above definition, we now present our case study, the mutual exclusion algorithm by Ricart Agrawala [14]. Our illustration shows that RA is a well-defined 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-ids* 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 "don't care" events with the symbol "*". Therefore, the resulting process actions for j are as follows.

RA

```
{ Request CS } ⟶
    if (¬hungry.j) then
        local-event(j , REQ_ts.j , True);
        hungry.j:=True;
        (∀k : k ≠ j : send(j , * , REQ_ts.j , False));

{ Receipt of REQ_ts.k } ⟶
    receive(j , * , REQ_ts.k , False);
    REPLY.j.k := REQ_ts.k;
    if (hungry.j ∧ ¬less-than (REQ_ts.k , REQ_ts.j))
    then deferred_set.j := deferred_set.j ∪ {k};
    else send(j , * , REPLY.j.k , False);

{ Receipt of REPLY.k } ⟶
    RECVD.j.k := REPLY.k;

hungry.j ∧ ∀k : k ≠ j : REQ_ts.j =RECVD.j.k ⟶
    CS.j := True;

{Release CS } ⟶
    (∀k : k ∈ deferred_set.j:
                    send(j , * , REPLY.j.k , False));
    deferred_set.j := ∅;
    hungry.j := False;
    CS.j := False
```

Note that RA is a well-defined 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 RA_D. Note that 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.

Case study (continued). The outline of the rest of the case study is as follows. In the next section, we demonstrate that RA satisfies a contract for using bounded-space RVC and, hence, a bounded-space RVC is substitutable for VC in RA. Likewise, in Section 5, we show that a bounded-space and stabilizing fault-tolerant RVC is substitutable for VC in RA. Based on this substitution and the method discussed in the Introduction, in Section 6 we show how to transform RA to be itself bounded-space and stabilizing fault-tolerant.

---

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, namely the substitutability of VC with bounded-space RVC.

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, say R, such that it will need to compare two events e and f only if $R(e, f)$ is true.

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 does an unbounded 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.

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.

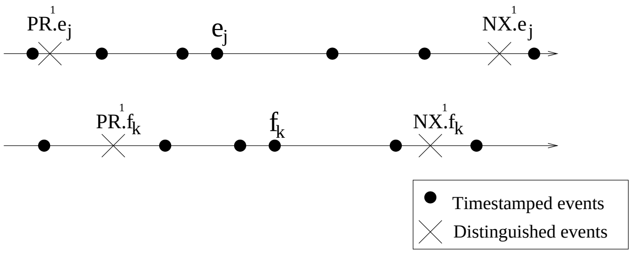

Figure 1: Illustration of PR^1 and NX^1 for events e_j and f_k

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

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

The client program can choose suitable values for m and n depending upon the comparison relation that is satisfied between events whose causality it needs to track. Note that if $m = \infty$ and $n = \infty$ then $R(m, n)$ is equal to true.

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. $\square$

Remarks. (1) For any two events e_1 and e_2, since "concurrent (e_1, e_2)" results in a call to happened-before(e_1, e_2) and happened-before(e_2, e_1), P_D guarantees both $R(m, n).e_1.e_2$ and $R(m, n).e_2.e_1$ to hold. (2) If a program satisfies $R(m, n)$, then it also satisfies $R(m', n')$ for any $m' \geq m$ and $n' \geq n$. $\square$

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 applications that do not satisfy $R(m, n)$ can be systematically modified to satisfy $R(m, n)$ by introducing periodic communication among processes. 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 RA_D satisfies the comparison predicate with $m=2$ and $n=1$.

Lemma 1. RA_D satisfies $R(2,1)$.

Proof. We introduce 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, RA_D satisfies $R(2,1)$. $\Box$

4.1.2 *Communication pattern*

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

$$
\begin{array}{l}
(\forall k, j, x : x \in N : \\
(1)\ (\exists e_k : (x^{th} \text{ distinguished event of } j\ \ \underline{hb}\ e_k)\ \wedge \\
\qquad \neg(x + M^{th} \text{ distinguished event of } j\ \ \underline{hb}\ e_k))\ \wedge \\
(2)\ (\forall m_k :: (\text{if } m_k \text{ is in transit at the time of} \\
\qquad\qquad j\text{'s } x^{th} \text{ distinguished event} \\
\qquad\quad \text{then } m_k \text{ is delivered before} \\
\qquad\qquad j\text{'s } x + M^{th} \text{ distinguished event}))\ \ )
\end{array}
$$

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

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, we have for any process k $ph.k.j$ is in the range $\{x - M\ \ldots\ x\}$ in S. Also, $ph.m.j$, for any message m that is in transit in S, is in the range $\{x - 2M\ \ldots\ x\}$.

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

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

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 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 RA_D satisfies the comparison predicate with $M=2$ and $l=2$.

Lemma 2. RA_D satisfies $comm(2,2)$.

Proof. 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 RA_D satisfies $comm(M,l)$ with $M = 2$ consider the two observations below.

(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 RA_D satisfies the communication pattern $comm(M,l)$ with $M = 2$. $\Box$

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 are similar to those of VC, except that the `increment-clock` is modified to use a bounded domain of values. 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 newer information regarding the phase about k. From the communication pattern $comm(M,l)$, $ph.m_l.k$ 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 newer 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, clock_bound )
 (∀j,k : j,k ∈ p_id_list : vc.j.k := 0, ph.j.k := 0);
 phase_bound := max(m + n − 1 , 3 * M + 1)

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 )
 (∀k : k ≠ j :
   if ( (ph.j.k < ph.m_l.k  ∧  ph.j.k + M + 1 > ph.m_l.k)
      ∨(ph.j.k > ph.m_l.k  ∧
         ph.j.k ≥ 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  ∧  vc.e_j.j ≤ vc.f_k.j)
   ∨  (ph.e_j.j < ph.f_k.j  ∧  ph.e_j.j + n > ph.f_k.j)
   ∨  (ph.e_j.j > ph.f_k.j  ∧  ph.e_j.j ≥ 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 inserts reset events when it executes a distinguished action. Thus, the value of $vc.k.j$ can never exceed l 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'' where the variable $ph'.e.j$ denotes the number of distinguished events of j that happened before the event e. 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. The following predicate is invariantly true in any computation of P_D.

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

The above 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, that is, whenever a process receives an "out of range message" with respect to the clock value of any other process, the local detection fires and 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 [4] 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. □

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 $ph.k.j$ 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 [5], we would like to note that it assumed for the stabilization of RVC 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. 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 a process + the maximum time a 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 action 1, and append **reset-timer(j)** to action 4 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. 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.

{ *timeout(j)* } $\longrightarrow$	*Release CS*(j); *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 stabilizing-tolerant RVC relies on its client to 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 in a deadlock situation, 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 the effects of a fault should be masked immediately), 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.

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. □

For reasons of space we relegate the proof to [5].

7. 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 (a) causality comparisons are performed for only a small subset of the events, and (b) comparisons are between events that occur within some number of adjacent "phases" of application execution. Observation (a) led to the simplification that only the events that are causally compared are freshly timestamped. Observation (b) led to the introduction of nonblocking resets; thus both enabled 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. Also, as discussed in Section 4, even if the application does not satisfy the contract it could be modified to do so.

Elsewhere [2], 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 [14] to be both bounded-space and stabilizing fault-tolerant. Elsewhere [5], we have also used the method to design an online, bounded-space and stabilizing fault-tolerant version of Garg-Chase's transient predicate detector [8]. 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 [1] 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.

8. REFERENCES

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

[2] A. Arora, S. S. Kulkarni, and M. Demirbas. Resettable vector clocks. Technical Report OSU-CISRS-2/00-TR05, Ohio State University, January 2000.

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

[4] 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.

[5] M. Demirbas. Resettable vector clocks: A case study in designing graybox fault-tolerance. Master's thesis, Ohio State University, February 2000. Also available as a technical report OSU-CISRC-4/00-TR11 from Department of Computer Information and Science, Ohio State University.

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

[7] 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.

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

[9] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *ACM Symposium on Principles of Distributed Computing*, 1996.

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

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

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

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

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

[15] 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.

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

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

$\square$