Resettable Vector Clocks: A Case Study in Designing Graybox Fault-Tolerance

By

Murat Demirbas, M.S.

The Ohio State University, 2000

Professor Anish Arora, Adviser

The task of designing fault-tolerance for large-scale applications (applications that inevitably contain multiple components) can be significantly simplified by designing fault-tolerance at the component level. In contrast to the traditional *whitebox* and *blackbox* methods, a *graybox* method for designing fault-tolerance to components allows the design of scalable and low-cost fault-tolerance by exploiting the contracts of the components.

In this thesis, we present a case study in designing graybox fault-tolerance. Our case study focuses on designing bounded-space fault-tolerance for applications that use vector clocks (VC). To this end, we generalize the notion of VC to resettable vector clocks (RVC) and identify a contract under which a bounded-space RVC implementation can be substituted for VC in client applications without affecting the client's correctness. Further, we design fault-tolerance to bounded-space RVC by exploiting its contract. Based on this bounded-space and fault-tolerant RVC, we

show how to design bounded-space fault-tolerance to VC applications; we demonstrate our method in the context of Ricart-Agrawala's mutual exclusion program and Garg-Chase's predicate detection program.

Resettable Vector Clocks: A Case Study in Designing Graybox Fault-Tolerance

By

Murat Demirbas, M.S.

The Ohio State University, 2000

Professor Anish Arora, Adviser

The task of designing fault-tolerance for large-scale applications (applications that inevitably contain multiple components) can be significantly simplified by designing fault-tolerance at the component level. In contrast to the traditional *whitebox* and *blackbox* methods, a *graybox* method for designing fault-tolerance to components allows the design of scalable and low-cost fault-tolerance by exploiting the contracts of the components.

In this thesis, we present a case study in designing graybox fault-tolerance. Our case study focuses on designing bounded-space fault-tolerance for applications that use vector clocks (VC). To this end, we generalize the notion of VC to resettable vector clocks (RVC) and identify a contract under which a bounded-space RVC implementation can be substituted for VC in client applications without affecting the client's correctness. Further, we design fault-tolerance to bounded-space RVC by exploiting its contract. Based on this bounded-space and fault-tolerant RVC, we

show how to design bounded-space fault-tolerance to VC applications; we demonstrate our method in the context of Ricart-Agrawala's mutual exclusion program and Garg-Chase's predicate detection program.

Resettable Vector Clocks: A Case Study in Designing Graybox Fault-Tolerance

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Murat Demirbas, B.S.

* * * * *

The Ohio State University

2000

Master's Examination Committee:

Professor Anish Arora, Adviser

Professor Paolo Sivilotti

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

The task of designing fault-tolerance for large-scale applications (applications that inevitably contain multiple components) can be significantly simplified by designing fault-tolerance at the component level. In contrast to the traditional *whitebox* and *blackbox* methods, a *graybox* method for designing fault-tolerance to components allows the design of scalable and low-cost fault-tolerance by exploiting the contracts of the components.

In this thesis, we present a case study in designing graybox fault-tolerance. Our case study focuses on designing bounded-space fault-tolerance for applications that use vector clocks (VC). To this end, we generalize the notion of VC to resettable vector clocks (RVC) and identify a contract under which a bounded-space RVC implementation can be substituted for VC in client applications without affecting the client's correctness. Further, we design fault-tolerance to bounded-space RVC by exploiting its contract. Based on this bounded-space and fault-tolerant RVC, we show how to design bounded-space fault-tolerance to VC applications; we demonstrate our method in the context of Ricart-Agrawala's mutual exclusion program and Garg-Chase's predicate detection program.

Dedicated to my parents (Ahmet and Perihan) and my brothers (Engin and Ersin) for their endless love.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank everyone who has ever helped me throughout my M.S. journey.

The most important among these is my advisor Prof. Anish Arora. His creativity, extraordinarily clear mind, and abstract thinking abilities have always amazed me. I am especially thankful to him for stimulating my interests for doing research and for his patience and kindness towards me. I will consider it a great accomplishment for me if I can acquire a little bit of his creativity, dedication, and perseverance throughout my Ph.D. study.

I am also very grateful to Prof. Sandeep Kulkarni of Michigan State university, who has been a perfect role model and a big brother for me during the last two years of his graduate study here in Ohio State. He helped me through technical and non-technical aspects of my study, and provided elegant answers to my endless questions patiently.

I would also like to thank Prof. Paolo Sivilotti, Rajesh K. Jaganathan, and Bill Leal for reviewing my drafts and helping me with my studies in their own way.

VITA

April 21, 1976 .... Born - Trabzon, Turkey

1997 .... B.S.
Computer Science and Engineering,
Middle East Technical University,
Ankara, Turkey.

Autumn98-present .... Graduate Research Associate,
Ohio State University.

PUBLICATIONS

Research Publications

A. Arora, S. S. Kulkarni, M. Demirbas. "Resettable vector clocks." *Technical report, Ohio State University*, January 2000.

M. Demirbas and A. Arora. "An optimal termination detection algorithm for rings." *Technical report, Ohio State University*, January 2000.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Fault-tolerance: Prof. Anish Arora

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Fault-tolerance is the ability of a system to deliver a desired level of functionality in the presence of faults. Fault-tolerance is crucial for many systems and is becoming vitally important for computing and communication based systems as they become irreplaceable tools of our modern society.

A striking example of this trend is the Internet, which has not only revolutionized the computer and communications world but has also impacted society deeply with the exponentially increasing use of online tools for electronic commerce, information acquisition, and entertainment. The recent (February 7-9, 2000) spate of cyber attacks toward "Yahoo.com", "CNN.com", and several e-commerce sites, a total of 9 sites, dramatically exposed that the Internet is still a work in progress and can be knocked down very easily. Also, the fact that these attacks did not require any sophisticated techniques suggests that a professionally organized attack may lead to a chaos.

In sum, despite our exponentially increasing demands and dependency on computing and communications, the software available today is not only far from being fault-tolerant but also away from responding the requests of the market. The growing distance between the demand and the supply for software pushes our society to the edge of a new software crisis.

Component software technology has been cited by many [30] as the cure for this software crisis, since it enables practical reuse of software "parts" and amortization of investments (both time and money) over multiple applications. Building new applications by composing components improves quality, shortens development time, and simplifies the maintenance of the application. Sun's Java Beans [29], Microsoft's DCOM [21], and OMG's CORBA [24] are three competing technologies of today that are designed to facilitate the development of component-based software.

1.1 Fault-Tolerant Components

The arguments that favor the component-based development of software also apply in the context of fault-tolerance. Therefore, rather than trying to design fault-tolerance for the entire application at once, it is more advantageous to first identify the components of the application and design fault-tolerance for each component, and then make the application fault-tolerant based on these fault-tolerant components. Since components are by definition smaller than the entire application, adding fault-tolerance to the components would be simpler than adding fault-tolerance to the entire application. Note, also, that making a component fault-tolerant would also simplify the design of fault-tolerance for another application which uses that component.

In sum, designing fault-tolerance for large-scale applications by composing fault-tolerant components enjoys all the advantages of component-based software development; it improves dependability, shortens development time, allows reuse, and simplifies both the addition and maintenance of fault-tolerance.

Although there is a lot of work [1, 2, 15] on designing fault-tolerance for small applications, there is little work [4, 7] on designing fault-tolerance for large-scale applications. There is an urging need for a method for designing scalable, reusable and easily maintainable fault-tolerance for large scale applications. As we argued above, transforming each component of the application to be fault-tolerant and composing them to achieve fault-tolerance for the entire application is one advantageous way to attack this problem.

1.2 Methods for Designing Fault-Tolerant Components

We classify the methods for designing fault-tolerance to components as being *whitebox*, *blackbox*, or *graybox* approaches.

A whitebox design assumes that the implementation of the component is fully available. Various whitebox methods for designing fault-tolerance have been proposed in the literature including exception handling, forward recovery, recovery blocks [25], and application-specific fault-tolerance methods [1, 2].

Whitebox designs of fault-tolerance allows low-cost (efficient in terms of space and time) solutions to be developed by exploiting the implementation details. However, whitebox design of fault-tolerance for components has several drawbacks. Firstly, this method is not scalable because the task of studying the implementation and designing a fault-tolerant version gets harder as the size of the implementation grows. Secondly, a whitebox view of a component is usually not possible, since the component vendors do not want the details of their implementation to be revealed. Also, since a whitebox design of fault-tolerance is implementation specific, the fault-tolerance designed by this approach is not reusable.

A blackbox design of fault-tolerance to a component does not assume any knowledge about the component except its interface. Some examples of blackbox methods for designing fault-tolerance are triple modular redundancy, quorum-based techniques, and primary-backup approaches. Blackbox designs of fault-tolerance have the serious drawback of being extremely high-cost solutions.

A graybox design of fault-tolerance to a component assumes that only the contract (the interface, the specifications, and possibly a controlled part of the implementation) of the component is available. The graybox design approach suggests studying the contract rather than the entire implementation, and hence, it is scalable and reusable in contrast to a whitebox approach. Also since the graybox approach assumes some knowledge about the contracts of the components, it allows the design of efficient fault-tolerance in contrast to a blackbox approach. Therefore, this approach avoids the drawbacks of both the whitebox and blackbox methods, and allows the design of scalable, reusable, and low-cost fault-tolerance.

While designing fault-tolerance for an intolerant component C, the graybox methods exploit the contract of C and add a *wrapper* code around C to transform it to be fault-tolerant. Graybox methods of designing fault-tolerance to a component allow reusable fault-tolerance in that the wrapper built for C also adds fault-tolerance to other implementations of C provided that they have the same contract as C.

Although the whitebox and blackbox approaches are studied extensively in the literature, there is no work on graybox designs of fault-tolerance.

1.3 Designing Graybox Fault-Tolerance for Component-Based Systems

As motivated in Section 1.1, in order to design fault-tolerance for large-scale applications, it is more advantageous to first design fault-tolerance at the component level. Also, as argued in Section 1.2, a graybox method for designing fault-tolerance to components allows the design of scalable and low cost fault-tolerance by exploiting the contracts between the components. Therefore, we outline a strategy for designing scalable and low cost fault-tolerance to large-scale applications as follows. Given a fault-intolerant large-scale application and a desired level of fault-tolerance that the application should satisfy, the first step should be to identify its components, and the contracts between them. The next step is to exploit the contracts between the components to design a suitable level of fault-tolerance to each component.

The final step is to design fault-tolerance to the entire application by using its fault-tolerant components as building blocks. To this end, we use a method that is based on the following observation: In any application that consists of only one client and the components it uses, faults occur at three levels: (1) in the client, (2) in the components, or (3) in the interface between a component and its client. Our method deals with these three levels separately, as follows. For level (1), the client 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 (2), the component self-reflectively invokes its own operations to ensure that it is in a consistent state. It may also notify the client about its state change, so the client may correct its state accordingly. For level (3), which may arise even when the client and the component are both internally consistent (but they are mutually inconsistent, i.e. their interface invariant is violated), detection is performed by the component or the client. Here, the method optimistically tries to deal with the fault by assuming that the client and the component are in internally consistent states. Therefore, if the component detects an interface fault, it notifies the client. The client upon receiving such notification 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.)

In the general case, a component used by the client may itself consist of smaller components that it uses. In this general case where we have an hierarchical design, the above method is still applicable within each client-component pair.

1.4 Thesis

In this thesis, we present a case study in designing graybox fault-tolerance. Our case study focuses on designing bounded-space fault-tolerance for applications that use vector clocks.

To this end, we first design bounded-space fault-tolerance for the vector clock component by using the graybox approach of exploiting its contract (its interface, specification, and a very limited part of its implementation). Then based on this fault-tolerant vector clock component, we show how to design bounded-space fault-tolerance to the applications that use vector clocks. More specifically, we demonstrate our method to design bounded-space fault-tolerance for Ricart-Agrawala's mutual exclusion program and Garg-Chase's predicate detection program. In both cases, to the best of our knowledge, prior solutions have lacked bounded-space and/or fault-tolerance.

1.4.1 Vector clock component

Vector clocks (VC) are an inherent component of a large class of distributed applications. This is due to the fact that distributed systems deal with their lack of a global time by using other abstractions of time and VC provides one such abstraction by capturing the causality relation between events in a distributed computation. VC, independently proposed by Fidge [12] and Mattern [20], is extensively used in distributed applications, such as, distributed debugging [13], checkpointing and recovery [16, 18], and causal communication [27].

Applications that use VC may require unbounded space, since vector clocks grow unboundedly. In order to 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 should 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 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.

In order to design fault-tolerance for VC applications, merely 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 [17] 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.

1.4.2 Designing graybox fault-tolerance for applications that use vector clocks

As motivated in the introduction and as the above observations show, in order to design fault-tolerance for VC applications, we should first make VC fault-tolerant. We call this problem of substituting VC with a bounded-space and fault-tolerant RVC without affecting application correctness as the *RVC problem*.

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. After identifying a contract for bounded-space RVC, we present a bounded-space RVC by building a wrapper around VC in a graybox manner.

Next, we design stabilizing fault-tolerance for bounded-space RVC by exploiting its contract.

Our goals in solving the RVC problem are as follows. The identified contract for bounded-space and fault-tolerant RVC 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 next step is to design fault-tolerance to VC applications using this fault-tolerant RVC as a building block. To this end, we use the method described in Section 1.3. We give two illustrations of our method; the application in the first illustration is Ricart and Agrawala's mutual exclusion [26], and in the second is Garg and Chase's transient predicate detector [13]. To the best of our knowledge, in both cases, prior solutions have lacked bounded-space and/or stabilizing-tolerance.

1.5 Outline of the Thesis

In Chapter 2, we discuss the system model and define the RVC problem. In Chapter 3, we present a bounded-space RVC by building a wrapper around VC in a graybox manner, and identify a contract that should be satisfied for the clients to use this bounded-space component. In Chapter 4, we design stabilizing fault-tolerance for the bounded-space RVC by exploiting its contract. In Chapters 5 and 6, we use the bounded-space and fault-tolerant RVC to design bounded-space and stabilizing fault-tolerant versions for the mutual exclusion solution by Ricart-Agrawala [26] and the transient predicate detector by Garg-Chase [13]. We make concluding remarks in Chapter 7. We relegate all proofs to the Appendix.

CHAPTER 2

PRELIMINARIES

2.1 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. Each action is of the form:

$$\langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$$

The guard of each action is a boolean expression over the program variables. The execution of the statement of each action is atomic and terminating, and updates 0 or more program variables and calls 0 or more component operations. Lamport's [19] 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.

2.2 Vector Clock Component

The values in the interface of VC consist of a vector $vc.j$ for each process j. The operations in the interface of VC and their informal specifications are as follows:

- `constructor( process_id_list)` This operation creates and initializes a vector clock component instance.

- `send( sender_id, event, message, flag )` This operation timestamps the message to be sent and the send event occurring in process `sender_id`. It also updates the clock values of process `sender_id` accordingly.

- `receive( receiver_id, event, message, flag )` This operation timestamps the receive event occurring in the process `receiver_id`. It also updates the clock values of process `receiver_id` based on the timestamp of the message received.

- `local-event( process_id, event, flag )` This operation timestamps the local events occurring in a process, and also updates the clock value of that process accordingly.

- `happened-before( event1, event2 ) : boolean` This operation compares the timestamps of the two events, and returns "true" if the first event happened before the other and "false" otherwise.

- `concurrent ( event1, event2 ) : boolean` This operation compares the timestamps of the two events, and returns "true" if they are concurrent and "false" otherwise.

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 thesis.

We give an implementation of VC in Figure 2.1. Note that this implementation 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 [6], 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.

2.2.1 Resettable vector clock component

The interface of RVC subsumes that of VC, however, it modifies the constructor to include a client contract (see Section 3.1) 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 required in the absence of faults, and invoked only upon the detection of a fault.

Vector Clock Component

```
constructor ( p_id_list )
    (∀j, k : j, k ∈ 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 )
    (∀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 ≤ vc.f_k.j)

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

Figure 2.1: VC implementation

2.3 The RVC problem

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 $\mathit{flag} = True$. □

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.

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 may compare.

2.4 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 [11].

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 $var.j$.

CHAPTER 3

BOUNDED-SPACE RESETTABLE VECTOR CLOCKS

In this chapter, we present a bounded implementation of RVC by building a wrapper around VC in a graybox manner, and identify a contract that should be satisfied for the clients to use this bounded-space 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.

3.1 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 time for which a message is delayed is also bounded; if a message m stays in a channel for a long enough time such that a message m' with identical timestamp can be created, then the receiving process will not be able to update its clock appropriately.

3.1.1 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.

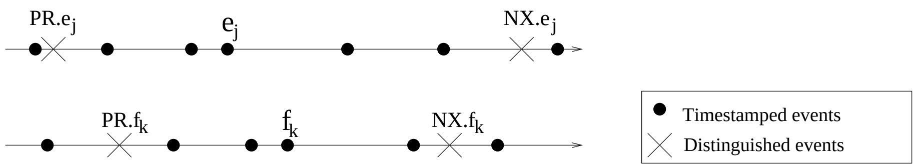

Figure 3.1: Illustration of PR^1 and NX^1 for events e_j and f_k

We define the comparison predicate $R(m, n)$ as follows:

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

Note that if $m = \infty$ and $n = \infty$ then $R(m, n)$ is equal to true. 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.

Definition. A program P 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. $\square$

Remark. 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. $\square$

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.

3.1.2 Communication pattern

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

$(\forall k, j, x : x \in N :$

(1) $(\exists e_k :$ $(x^{th}$ distinguished event of j $\underline{hb}$ $e_k)$

$\wedge \neg(x + M^{th}$ distinguished event of j $\underline{hb}$ $e_k))$

$\wedge$

(2) $(\forall m_k ::$ (if m_k is in transit at the time of j's x^{th} distinguished event

then m_k is delivered before j's $x + M^{th}$ distinguished event))))

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, $l \in Z^+$. □

As is the case with the comparison predicate, a large class of applications satisfy $comm(M, l)$ for some M and l. For example, phase-based applications where each phase uses a standard communication pattern such as a diffusing computation over FIFO channels or heartbeat messages over time bounded channels satisfy *comm*. Even the applications, where an unbounded number of messages might be send at each phase (to ensure that the message will be delivered eventually), can be modified to satisfy $comm(M, l)$, since we can find a bounded l by freshly timestamping only the first message send event, and using the current timestamp for all the repetitions of the message.

3.2 Bounded-Space Resettable Vector Clock Component

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 *clock_bound* to l. 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. The correctness of the `receive` operation depends upon $comm(M, l)$ and that of `happened-before` depends on $R(m, n)$. The `nonblocking-reset`(j) sets $vc.j.j$ to 0 and increments the $ph.j.j$ by 1 modulo *phase_bound*.

While designing the wrapper in a graybox manner, other than the interface and the specification for VC, we assume access to the VC entries $(vc.j.k)$ in order to assign them to other VC entries. We also assume *max(x,y)* operation which returns the maximum of the two VC entries, and *reset-local-clock(j)* operation which sets $vc.j.j$ to its initial value. In the implementation, we write **VC::X** to denote an operation X on VC. Thus, our bounded-space RVC implementation is as shown in Figure 3.2.

Theorem 1. *(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. □

Bounded-Space Resettable Vector Clock Component

```
constructor ( p_id_list, R(m,n), M, clock_bound )
   VC::constructor ( p_id_list);
   (∀j,k : j,k ∈ p_id_list : ph.j.k := 0);
   phase_bound := max(m + n − 1 , 3 ∗ M + 1);

send ( j, e_j, m_j, flag )
   VC::send ( j, e_j, m_j, flag );
   ph.e_j, ph.m_j := ph.j, ph.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::vc.j.k := ph.m_l.k, VC::vc.m_l.k
      elseif (ph.j.k = ph.m_l.k) then VC::vc.j.k := VC::max(vc.j.k, vc.m_l.k)
      else   /* don't update ph.j.k and vc.j.k */  );
   VC::local-event ( j, e_j, flag );
   ph.e_j := ph.j;

local-event ( j, e_j, flag )
   VC::local-event ( j, e_j, flag );
   ph.e_j := ph.j;

happened-before ( e_j, f_k ) :  boolean
   return (      (ph.e_j.j = ph.f_k.j  ∧ VC::happened-before ( e_j, f_k ) )
            ∨    (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::increment-clock ( j );

nonblocking-reset ( j )
   ph.j.j := ph.j.j + 1 mod (phase_bound);
   VC::reset-local-clock(j);
```

Figure 3.2: Bounded-space RVC implementation

CHAPTER 4

STABILIZATION OF RESETTABLE VECTOR CLOCKS

In this chapter, we design stabilizing fault-tolerance for bounded space RVC by exploiting its contract. More specifically, we design a wrapper around the bounded-space RVC (in a graybox manner) in order to transform it to be stabilizing-tolerant also.

In order to design stabilizing fault-tolerance for RVC, we assume that the client eventually starts re-satisfying its contract with bounded-space RVC even though the contract might be temporarily violated. We exploit the communication pattern part of the contract to design a local detector in order to detect any inconsistencies in RVC. Recall that if the communication pattern holds then for every message m, $ph.m.j$ will be in the range $\{ph.j.j - 2M \ \ldots \ ph.j.j\}$ according to any j. RVC locally detects the out of range messages and calls the "global-reset" operation self-reflectively whenever a process receives an out of range message.

Remark. We could have alternatively used superposed snapshots to globally detect a violation of the following predicate of bounded RVC; $(\forall j, k :: (ph.j.j -_{p\_b} M) <_{p\_b} ph.k.j <_{p\_b} ph.j.j)$, where X_{p_b} denotes operation X in modulo *phase_bound* arithmetic. However, we rule out this solution for the sake of efficiency since it generates extra messages.

Local detection. The following predicate is invariantly true in any computation of a client of bounded-space RVC.

$$(\forall j,k : j \neq k : ((ph.j.j \ -_{p\_b} \ 2M) \ <_{p\_b} \ ph.m.j \ <_{p\_b} \ ph.j.j)$$
$$\wedge \ ((ph.j.k \ -_{p\_b} \ 2M) \ <_{p\_b} \ ph.m.k \ <_{p\_b} \ (ph.j.k \ +_{p\_b} \ M)))$$

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 self-reflectively. 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 exists a cycle among the processes which causes them 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 N is the number of processes, E is the number of channels, B is the bound on the channel capacity, and M, m, and n are contract parameters as before.

Theorem 2. The modified bounded RVC is stabilizing fault-tolerant. □

The proof for stabilization of this modified component 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. The detailed proof is included in the Appendix.

Note that the proof assumed for the stabilization of the modified component that the application perform enough number of resets. Therefore we identify this condition as an additional contract to be satisfied by the application in order for the modified component to recover from faults.

Theorem 3. *(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. □

CHAPTER 5

APPLICATION: STABILIZING RICART-AGRAWALA MUTUAL EXCLUSION

In this chapter, we demonstrate the method discussed in Section 1.3 on the Ricart-Agrawala program. In Section 5.1, we present a version of Ricart-Agrawala mutual exclusion program that uses VC, namely *RA*, and then in Section 5.2 we show that VC is substitutable with bounded RVC in RA. In Section 5.3, we substitute VC with stabilizing and bounded RVC, and design a wrapper around the bounded RA by using the graybox method described in Section 1.3 to transform the RA program to be stabilizing fault-tolerant. Finally, in Section 5.4, we discuss the reusability of the wrapper designed in Section 5.3 for Lamport's mutual exclusion program [19].

5.1 RA program

In this section, we present a version of Ricart-Agrawala mutual exclusion program that uses VC, namely *RA*. 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.

Symbols	Explanation
REQ_ts.j	the timestamp of the last *REQUEST* at j.
deferred_set.j	the set of processes that j has deferred to send a *REPLY*.
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 denoting whether or not j is currently waiting for CS.

Table 5.1: Program variables of RA

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 `)` $\vee$ (`concurrent(` e_j , f_k `)` $\wedge$ ($j < k$)));

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 "*". We give an implementation of RA in Figure 5.1.

RA program for process j

$\{$ *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
  $deferred\_set.j := deferred\_set.j\ +\{k\}$;
 else **send**$(j\ ,\ *\ ,\ REPLY.j.k\ ,\ False)$;

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

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

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

Figure 5.1: RA program

5.2 Bounding the space of RA

In this section we bound RA in a graybox manner (by exploiting its contract). In order to bound RA we substitute VC with a bounded-space RVC. To this end, we first "reset annotate" RA. From the specification of RA, we know that processes loop through the following three actions; *Request CS*, *CS granted* (action 4), and *Release CS*. Since each loop execution can be seen as a new phase in the computation of a process, we can select any of these three actions as the distinguished action of RA. We choose to identify the *Release CS* action as the distinguished action of RA, and put a *wrapper* around the *Release CS* action of j that appends a call to `nonblocking-reset` operation for j. We now identify the communication pattern and the comparison predicate RA satisfies as follows.

Lemma 4. Modified RA satisfies the comparison predicate $R(2,1)$. □

Lemma 5. Modified RA satisfies $comm(M,l)$ with $M=2$, and $l=2$. □

Theorem 6. A bounded-space RVC with the constructor $(p\_ids, R(3,2), 2, 2)$ is substitutable for VC in modified RA. □

Note that, the *phase_bound* of C' is $max(3+2-1\ ,\ 3*2+1) = 7$. Therefore, modified RA uses only bounded space.

5.3 Stabilization of Ricart-Agrawala

In this section, we use the method described in Section 1.3 to design stabilizing-tolerance to the RA application which consists of VC and its client (main module in RA).

5.3.1 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 ((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. To this end, we put a wrapper around the main module that starts the timer for a process j whenever j calls a *Request CS* action, and resets j's timer whenever j is *granted the CS*.

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, in order to avoid this high-cost solution we choose to exploit the contract of the main module further. 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, since 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 piece of code to the wrapper for the main module.

{ *timeout(j)* } $\longrightarrow$ *Release CS*(j); *Request CS*(j);

5.3.2 Stabilization of VC

By using Theorem 3 we substitute VC with a bounded-space and stabilizing-tolerant RVC. Note from the contract for the stabilizing-tolerant RVC that, 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 (*phase_bound*) of reset events at every process. Also if RVC detects an inconsistency and invokes its `global-reset` operation self-reflectively, it notifies the main module about this decision which in turn releases the CS for all processes in order to start fresh and synchronized with RVC.

5.3.3 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 the RVC. It may be the case that both the main module and RVC are internally consistent but the above condition is false, because the requests by processes 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 the inconsistencies at the interface invariant will be corrected eventually. It is indeed the case for the wrapper we designed in Sections 5.3.1 and 5.3.2. It may be the case that incorrect request values are detected by the RVC since they are out of the valid range for the `receive` operation. In this case, the application (hence the internal invariant) stabilizes as discussed in Section 5.3.2. Otherwise, the interface invariant stabilizes after all the processes requests for new CS executions, since the new requests will get a correct value from RVC.

We assumed without loss of generality that both RVC and the main module has been stabilized, since otherwise Sections 5.3.1 and 5.3.2 would apply.

Theorem 7. RA with the wrapper designed in Sections 5.3.1, 5.3.2, and 5.3.3 is stabilizing fault-tolerant. □

5.4 Reusability of the Wrapper for Lamport's Mutual Exclusion

In this section, we discuss the reusability of the wrapper designed in Section 5.3 for Lamport's mutual exclusion program [19]. Note that we designed the wrapper by using a graybox method of designing fault-tolerance, i.e., we only looked at the specification for Ricart-Agrawala rather than its implementation. Since Lamport's mutual exclusion program has the same specification as the Ricart-Agrawala program, the same wrapper can be reused to design stabilizing fault-tolerance to Lamport's program.

The remaining part is to investigate the Lamport's program to identify the communication pattern and the comparison predicate it satisfies in order to substitute VC with a bounded-space fault-tolerant RVC in a version of Lamport's program that uses VC. It turns out that Lamport's program satisfies $R(2,1)$ and can be modified to satisfy $comm(2,2)$ by not timestamping the $RELEASE$ messages.

Therefore, we can design a bounded-space and stabilizing-tolerance implementation of Lamport's mutual exclusion by just reusing the fault-tolerance argument for Ricart-Agrawala's mutual exclusion.

CHAPTER 6

APPLICATION: STABILIZING ONLINE PREDICATE DETECTOR

In this chapter, we demonstrate the method discussed in Section 1.3 on a conjunctive predicate detection program. First, in Section 6.1, we discuss the weak conjunctive predicate detection problem. In Section 6.2, we present Garg and Chase's weak conjunctive predicate detector program [13]. Then, in Section 6.3 , 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 in a graybox manner. Finally, in Section 6.4, we show how to make the detector itself stabilizing by using the graybox method described in Section 1.3.

6.1 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 [9] and unstable predicate detection problems [13, 14] have been considered in detail in the literature.

In this chapter, we focus on the detection of weak conjunctive predicates [14], i.e., transient predicates that are the conjunction of local predicates on a number of processes. 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 [14]. Our detector differs from that in [13, 14] 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.

6.2 A Weak Conjunctive Predicate Detector

In this section, we present a weak conjunctive detector due to Garg and Chase [13] 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 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 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 $vc.j.k$. 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, 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 as shown in Figure 6.1.

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);   // *sends the token to the next process on the ring*

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

Figure 6.1: A weak conjunctive predicate detector implementation

6.3 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 the client satisfies $comm(M, l)$ with $M = 1$ and for some l and 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, then j places the timestamp of the event which is just before this reset event to the events queue of j. 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 an event to its event queue with the same timestamp as 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 events to its queue.

Lemma 8. For any events e_j and f_k, if the weak conjunctive predicate detector compares e_j and f_k then $R(3,2).e_j.f_k$ is true. □

Corollary. From Theorem 1, 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.

Observe that 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 have dequeued 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 unless it can be determined that no such consistent cut can be found.

While designing the wrapper in a graybox manner, we assume access to 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 **predicatedetector::X** to denote an operation X on the predicate detector program presented in Section 6.2. Thus, our bounded-space predicate detector program is as shown in Figure 6.2.

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 Δ, $\Delta \geq 1$.

Bounded-space weak conjunctive predicate detector

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

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

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

Figure 6.2: A bounded-space weak conjunctive predicate detector implementation

6.4 Stabilizing Online Predicate Detector

In this section, we use the method described in Section 1.3 to design stabilizing-tolerance to the online predicate detector application which consists of VC and its client (main module in predicate detector).

6.4.1 Stabilization of the main module

In order to design stabilizing-tolerance to the main module of the predicate detector, we exploit its specification. 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 Δ, 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 Δ+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 Δ 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 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. This check is to ensure that whenever process j compares two events e and f $R(3,2).e.f$ is satisfied.

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.

While designing the wrapper we assumed access to only the variable G of the predicate detector, and ph variable of RVC.

6.4.2 Stabilization of VC

By using Theorem 3 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.

6.4.3 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.

6.5 Proof Sketch

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

- Starting from an arbitrary state, as shown in [10], 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 $x-2, x-1, x, orx+1$. 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.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this chapter, we first summarize the important results in this thesis in Section 7.1, and then discuss the related work in Section 7.2. Finally, in Section 7.3, we outline possible extensions to this work.

7.1 Concluding Remarks

Broadly speaking, this thesis 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 components, 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 graybox method of designing fault-tolerance to large-scale applications, and demonstrated this method for designing bounded-space fault-tolerance to the applications that use VC.

To this end, 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 vector 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 component. 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 whether the contract is satisfied and what the parameter values are.

Elsewhere [3], 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 process 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 three different levels: in the component, the application, or the interface between the two. We demonstrated a method that deals with these levels, in terms of two applications: (1) Ricart-Agrawala's mutual exclusion program [26], and (2) Garg-Chase's predicate detector [13]. For each, we showed how the application is made both bounded space and stabilizing fault-tolerant. Prior solutions had lacked one or both of these properties.

7.2 Related Work

Related work about designing fault-tolerance for component-based applications. Stark presents a proof technique for rely/guarantee properties in [28]. A rely condition for a program P, denoted as R, expresses the conditions that P relies on its environment to provide, and a guarantee condition, denoted as G, expresses what P guarantees to provide in return. Stark presents a proof technique that permits us to infer that a program P satisfies a rely guarantee specification $R \Longrightarrow G$, given that we know P satisfies a finite collection of rely/guarantee specifications $R_i \Longrightarrow G_i$. This proof technique can be applied to the component based applications simply by denoting the rely/guarantee specification of the entire application as $R \Longrightarrow G$, and the rely/guarantee specification of each component i as $R_i \Longrightarrow G_i$. However, when faults are concerned things get complicated, since components will now have an ideal specification and a fault-tolerance specification. The proof technique should be modified in order to be applicable to the verification of fault-tolerance.

Although Cau and deRover [8] applied Stark's proof technique for verification of fault-tolerance using relative refinement, Stark's proof technique has not yet been considered for verification of fault-tolerance for component based applications.

Related work about vector clocks. Mostefaoui and Theel [23] 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 [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, in addition to enabling application stabilization via RVC stabilization, we also 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.

In contrast to our mutual exclusion program, the original version due to Ricart and Agrawala [26] uses $O(log\ n)$ space per process, where n is the number of processes, and is not stabilizing fault-tolerant. Nesterenko [22] gave a stabilizing version, but his solution uses unbounded state. Our solution uses $O(n\ \ log\ n)$ space per process.

In contrast to our transient predicate detection program, the original version due to Garg and Chase [13] is not online, bounded space or stabilizing fault-tolerant.

7.3 Future Directions

Our work suggest 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 to—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-tolerance for large scale applications, there is still a need for formal methodologies in this context. One possible direction for future work is the study of formal methods and component frameworks for designing graybox fault-tolerance for various classes of tolerance specification. An intermediate step toward this direction is to investigate proof methods that exploit the client-component contract to prove substitutability of a component by a fault-tolerant version of that component.

Another possible direction for future work is the study of scalable techniques for verification of fault-tolerance in large scale applications. This work could built upon Stark's proof technique in order to verify that a certain fault-tolerance property holds in the system by just looking at the fault-tolerance properties of its components.

APPENDIX A

PROOFS

Appendix A1: Proof of Theorem 1.

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 C' to update the clock values correctly when P annotated with D uses C'. Then, in the second part, we show that that condition is also sufficient for C' to keep track of the causality relation correctly when it is used by P annotated with 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 C' returns the same result as "happened-before(e_j, f_k)" of C. From the definition of substitutability in Chapter 2, it follows that C' *is substitutable for* C *in* P.

1) Note that, when using C', P annotated with 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 will still update $ph.k.j$ in the same way as in the case of the nonresettable vector clock.

To show this, we introduce an unbounded auxiliary variable, ph' such that $ph'.k.j$ denotes the the number of (events corresponding to the) distinguished actions 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 these two 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 on page 23, 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. ($PR^m.e_j \ \underline{hb} \ f_k \quad \wedge \neg(NX^n.e_j \ \underline{hb} \ f_k)$) 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 (events corresponding to the) distinguished actions 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 $(e_j \ \underline{hb} \ f_k) \ \equiv \quad ph'.e_j.j < ph'.f_k.j \ \vee \ (ph'.e_j.j = ph'.f_k.j \wedge vc.e_j.j \leq vc.f_k.j)$

Note that the bounded phase variable $ph.j.k$ of C' 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 C' 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 C' 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 C'.

  - 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 C'.

□

Appendix A2: Proof of Theorem 2.

Proof. 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 23, 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. $\square$

Appendix A3: Proof of Theorem 3.

Proof. This proof follows trivially from Theorems 1 and 2. $\square$

Appendix A4: Proof of Lemma 4.

Proof. The Ricart-Agrawala program compares two events e_j, f_k by the "happened-before" operation iff e_j is $REQ\_ts.j$ and f_k is $REQ\_ts.k$.

Similar to the proof of Theorem 1, consider an unbounded auxiliary variable ph'. Let e_j be the event denoting the request of j. Observe that $ph'.e_j.j$ denotes how often j has completed its critical section 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, f_k at process 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 that j completed more critical sections than that j believes.

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 on page 20, RA satisfies $R(2, 1)$. $\square$

Appendix A5: Proof of Lemma 5

Proof. Process j does its x^{th} reset when it completes the critical section for x^{th} time. Before j does the $(x+1)^{th}$ reset to complete critical section for the $(x+1)^{th}$ time, j sends a request message and receives a reply for the same. The x^{th} reset at j happened before the event created by the reception of the request message, and the event of the reception of the request message happened before the $(x+1)^{th}$ reset of j.

Let m be a message in transit from i to k when j does x^{th} reset. If m is a request message, process 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 does $(x+2)^{th}$ reset, 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 unless i also 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 other words, if m is in transit when j does x^{th} reset, then it is delivered by the time j does $(x+2)^{th}$ reset. Also, observe that there is only one event, the request of j, is freshly timestamped between two resets of j. It follows that the RA satisfies the communication pattern $comm(2,2)$. $\square$

Appendix A6: Proof of Theorem 7

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

$$\begin{array}{l}
I_1 = (\forall j,k : j \neq k : (j \in deferred_set.k) \Longrightarrow \text{less-than}(REQ_ts.k\ ,\ REQ_ts.j)) \\
I_2 = (\forall j,k : j \neq k : REPLY.j.k = \text{last } REQ_ts.k \text{ that } j \text{ has received from } k) \\
I_3 = (\forall j : CS.j \Longrightarrow (\forall k : k \neq j : REQ_ts.j = RECVD.j.k)) \\
I_4 = (\forall j : hungry.j \Longrightarrow REQ_ts.j.j = \text{RVC}.j.j) \\
I_5 = (\forall j,k : j \neq k : (j\text{'s request})\ \ \underline{hb}\ (k\text{'s request}) \\
\qquad\qquad\qquad\qquad \Longrightarrow \text{happened-before}(REQ_ts.j\ ,\ REQ_ts.k)\text{=True }) \\
I = I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5
\end{array}$$

From the timeout action, observe that each process will invoke the non-blocking reset operation unbounded number of times. From the proof of Theorem 2, 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. □

Appendix A7: Proof of Lemma 8

Proof. Observe that our detector guarantees the following property: Any event f_k at process k (see Figure A.1) 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).

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 :

$(ph.e_j.j \geq ph.f_k.j - 1 \ \vee \ ph.e_j.j \leq ph.f_k.j + 2) \ \wedge \ (ph.f_k.k \geq ph.e_j.k - 1 \ \vee \ ph.f_k.k \leq ph.e_j.k + 2)$

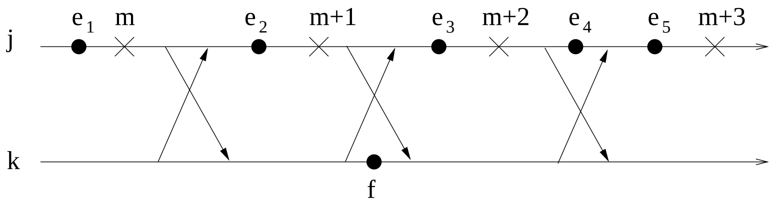

Figure A.1: The comparisons required by the detector

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. $\square$

APPENDIX B

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
$\wedge$, lor	conjunction, disjunction
$\Longrightarrow$, $\Longleftarrow$	implication, consequence
$\equiv$, $\not\equiv$	equivalance, inequivalence

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

BIBLIOGRAPHY

[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 and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper has been invited to *IEEE Transactions on Computers*.

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

[4] A. Arora and D. Poduska. A timing-based schema for stabilizing information exchange in networks. *Proceedings of the Third International Conference on Computer Networks*, 1995.

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

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *"ACM Transactions on Computer Systems"*, 9(3):272–314, 1991.

[7] K. P. Birman and R. van Renesse. *Reliable distributed computing using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[8] Antonio Cau and Willem-Paul de Roever. Specifying fault tolerance within Stark's formalism. In *Proc. of the Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[16] Ö. Babaoğlu and K. Marzullo. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms.* In Sape Mullender, editor, *Distributed Systems*, pages 55–96. Addison Wesley, New York, NY, 2nd edition, 1994.

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

[18] R. Koo and S. Toueg. Checkpointing and rollback-recovery in distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, Jan 1987.

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

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

[21] Microsoft. *DCOM.* http://www.microsoft.com/com/tech/DCOM.asp.

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

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

[24] OMG. *CORBA.* http://www.omg.org.

[25] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering.*, pages 220–232, 1975.

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

[27] 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.

[28] E. W. Stark. A proof technique for rely/guarantee properties. *Foundations of software technology and theoretical computer science, Lecture Notes in computer science 106*, pages 369–391, 1985.

[29] Sun. *Java Beans.* http://java.sun.com/products/ejb/.

[30] C. Szyperski. *Component Software.* Addison-Wesley, 1997.