Virtually-Synchronous Communication Based on a Weak Failure Suspector

André Schiper, Aleta Ricciardi*

Department of Computer Science, Upson Hall
Cornell University
Ithaca, NY 14853-7501

Abstract

Failure detectors (or, more accurately Failure Suspectors *– FS) appear to be a fundamental service upon which to build fault-tolerant, distributed applications. This paper shows that a FS with very weak semantics (i.e. that delivers failure and recovery information in no specific order) suffices to implement* virtually-synchronous communication *(VSC) in an asynchronous system subject to process crash failures and network partitions. The VSC paradigm is particularly useful in asynchronous systems and greatly simplifies building fault-tolerant applications that mask failures by replicating processes. We suggest a three-component architecture to implement virtually-synchronous communication : 1) at the lowest level, the FS component; on top of it, 2a) a component that defines new views, and 2b) a component that reliably multicasts messages within a view.*

1 Introduction

There have recently been several papers about membership services in asynchronous systems [2, 12, 16, 17, 18]. A membership service is responsible for giving each process (consistent) information about the operational processes in the system. A process calls this information its *view* of the system processes. A membership service typically reacts to process crashes or recoveries, leading it to define a set of views. The membership services mentioned vary according to the underlying failure model considered, as well as the properties they provide with respect to the set of views delivered to each process: (e.g. whether another view may exist simultaneously, the degree of agreement among members):

- [16, 17] consider processes with crash failure semantics, excluding network partitions.
- [18] considers systems in which processes may crash and the network may partition. However, despite network partitions, this membership service defines only *majority views* – a unique, totally-ordered sequence of views. Such a membership service is said to have *linear* semantics.
- The membership services described in [1, 2, 12] consider the same failure scenario as above, but only define a partial order on the views. That is, if the system is partitioned in two (or more) subnetworks then two (or more) views, one in each subnetwork, may exist concurrently.

Concurrent views offer an interesting extension to membership services, and force us to consider a further semantic distinction based on whether concurrent views are permitted to intersect. If two concurrent views may overlap, we say the membership service semantics are *weak-partial*, if they may not we say the semantics are *strong-partial*. Among those that permit concurrent views, [2] appears to be a strong-partial membership service. [12] considers both strong-partial and weak-partial membership services, and [1] considers only weak-partial membership service. These variants raise a new, pertinent question: when is a strong-partial service required, and when does a weak-partial membership service suffice. The objective of this paper is to suggest an answer to this question, by showing that a strong-partial membership service is intimately related to *virtually-synchronous* communi-

*The first author is on leave from Ecole Polytechnique Fédérale de Lausanne, Switzerland. His research is supported by the "Fonds national suisse" under contract number 21-32210.91, as part of the European ESPRIT Basic Research Project Number 6360 (BROADCAST).The second is supported by DARPA/NASA Ames Grant NAG 2-593, and by grants from IBM and Siemens Corporation.

cation. We do not discuss when a linear membership service is required.

The idea of virtually-synchronous communication (VSC) was first introduced by Isis [3, 4]. VSC can be understood as rule for ordering message deliveries (reliable multicasts) with respect to view changes (received from the membership service). We give a precise definition for VSC in Section 5.4. VSC defines a powerful model for building fault-tolerant processes that mask failures by replication. It has also been argued [5] that ordering message deliveries consistently around process failures and recoveries is a fundamental part of any distributed computation; thus VSC is a vital primitive for inherently-distributed programming. Relatedly, many common distributed applications are more easily understood and solved if they can make use of VSC [19]. Finally, if the VSC abstraction we define in this paper is augmented with a majority requirement, [20] shows it is a powerful model in which transaction commit is easily (albeit probabilistically) implemented. Understanding that the VSC abstraction is more basic than the transaction abstraction gives broader insight to the problem of building fault-tolerant applications. However, we note that solving VSC is not equivalent to solving consensus [10].

Traditionally virtually-synchronous communication has been implemented with a two component architecture: a membership service, and on top of it, multicast component. However, understanding the relationship between a membership service and virtually-synchronous communication has lead us to consider a three-component architecture, with (1) a *Failure Suspector* component FS delivering information about the communication topology, (2) a *View Component* VC defining *views*, and (3) a *Multicast Component* MC implementing virtually-synchronous communication. We divide the functionality of the traditional membership service between our FS and VC components.

In addition to increasing our understanding of the relationship between any membership service and virtually-synchronous communication, this architecture allowed us to specify precisely the FS semantics needed to guarantee VC and MC liveness. One weakness of previous work in this area has been a lack of precise semantics for the FS part of the system.

Explicitly, the paper shows:

- that virtually-synchronous communication satisfying the definition given in Section 5.4 can be implemented with a modular, three-component architecture for system models with both process crash failures and network partitions (i.e. link failures). We start with a very simple model, and from it construct a useful communication primitive for fault-tolerant, distributed applications.
- how to define concurrent views that have empty intersections. That is, how to implement strong-partial membership semantics in a system that may partition. The basic idea is to define a view as a set of pairs *(proc id, proc sequence number)*.
- that if we remove the MC component from the architecture (e.g. if virtually-synchronous communication is not needed), then the view component defines views that do not satisfy the empty intersection condition (i.e. giving a membership service with a weak-partial semantics).

Section 2 describes our low-level system model and the interaction of the three components. Section 3 gives a precise semantics for the failure suspector. Sections 4 and 5 sketch how to implement the VC_p and MC_p components, and Section 6 completes the VC_p and MC_p protocols. We conclude in Section 7.

2 System Model

Our low-level system model consists of an infinite name space of process identifiers, $\mathsf{Proc} = \{p_1, p_2, ..., \}$. The name space is infinite to model infinite executions in which processes continually fail and recover. At any point in time, however, there are only a finite number of executing processes under consideration and we restrict our attention to these. For this finite set of executing processes, we assume a completely-connected network of FIFO channels. Processes communicate by passing messages over these channels, though they too may fail. The system has no global clock, and message transmission delays are unbounded. Processes fail by crashing, which we model by the local event $crash_p$. We model the recovery of a process with a new identifier. A process p may (1) send a message to another process, (2) deliver a message sent by another process q, and (3) perform local computation.

A *history*, h_p, for process p is a sequence of events beginning with the event $start_p$ and terminating, if at all, with the event $crash_p$: $h_p = start_p \cdot e_p^1 \cdots e_p^k$, for $0 \leq k$. A *cut* is an n-tuple of process histories, one for each $p \in \mathsf{Proc}$. We assume familiarity with inter-event causality [14] and with consistent cuts [8].

Crash failures are surprisingly difficult to handle in an asynchronous system. Fischer, et.al [10] show that, because it is impossible to distinguish a crashed process from one that is just very slow, any problem requiring "all correct processes" to agree on some value cannot be solved deterministically; that is, no deterministic protocol can make progress if it must also make accurate process failure detections. One way around this is for asynchronous systems to incorporate some mechanism for *suspecting* failures, as well as a means of handling failure suspicions consistently (e.g. p may suspect q faulty while r may not; perhaps r and/or q even suspect p). Our system model assumes a *failure suspector* that eventually suspects a crashed process,[1] which suffices to ensure our protocols make progress. We do not require anything more of the failure suspector.

Each process has three components that interact to implement the virtually-synchronous communication primitive for application-layer processes (Figure 1). The *Failure Suspector* (FS_p) is at the lowest level and notifies both the *Multicast Component* (MC_p), and the *View Component* (VC_p) about suspected changes in the communication topology. Such changes arise from actual process and link failures, as well as high processor loads and heavy network traffic (indistinguishable from true failures). VC_p defines p's current *view*, $\mathsf{View}_p()$, an approximation of the set of processes with which p can communicate, and sends $\mathsf{View}_p()$ to MC_p. MC_p is responsible for reliably multicasting application-layer messages until it receives an accessibility-change notification from FS_p. These notifications signal a suspected change in the communication topology and the attendant need to alter $\mathsf{View}_p()$. However, neither MC_p nor VC_p can do this naively since virtually-synchronous communication requires that members of $\mathsf{View}_p()$ that also accompany p to its next view receive the same set of messages that were multicast within $\mathsf{View}_p()$ (We make this definition precise in Section 5). To ensure this, MC_p delivers all outstanding multicasts, and does not issue new multicasts except to forward those that have been only partially delivered. $\mathsf{View}_p()$ is safely terminated when all messages multicast in it are delivered at all sites that MC_p believes non-faulty. When MC_p detects this condition (Section 4) it informs VC_p, which then determines a new view for MC_p from the accessibility notifications it received from FS_p.

Section 3 describes the properties our Failure Suspector components must satisfy. These are weak yet reasonable requirements, and are easily implemented in any asynchronous system. Section 4 discusses VC_p, and Section 5 discusses MC_p. These components execute protocols to detect global properties [8, 15].

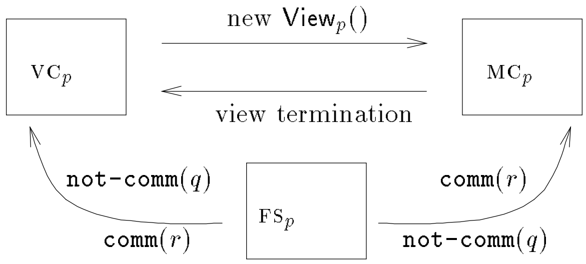

Figure 1: FS_p, MC_p, and VC_p interaction for virtually-synchronous communication.

3 The Failure Suspector

Given process p, FS_p emits a sequence of $\mathtt{not\text{-}comm}(q)$ and $\mathtt{comm}(r)$ suspicion messages to MC_p and VC_p. Since the system is asynchronous we cannot guarantee the accuracy or timeliness of these suspicions; the most we can require is that FS_p eventually suspects true crashes and recoveries. This is not unreasonable. It is known that fault-tolerant protocols in asynchronous systems cannot make progress if they are required to make accurate failure determinations. Our approach introduces an inaccurate failure suspector to gain liveness. On the other hand, we cannot require FS_p to suspect all periods of transient inaccessibility – a network partition may repair before it is noticed.

Since, in theory, FS_p may suspect processes arbitrarily, we have divorced FS_p implementation from the problem at hand. In a real system, FS_p might take cues from the underlying communication layer, the operating system, response delays, and so forth.[2]

On every consistent cut c, FS_p maintains two non-intersecting sets, $\mathsf{CommSet}_p(c)$ and $\mathsf{NotCommSet}_p(c)$. When FS_p suspects $q \in \mathsf{CommSet}_p(c)$, q is removed from $\mathsf{CommSet}_p(c)$ and is thereafter a member of $\mathsf{NotCommSet}_p(c)$. Whenever these sets change, FS_p notifies VC_p and MC_p by emitting the appropriate $\mathtt{comm}()$ or $\mathtt{not\text{-}comm}()$ messages.

We have a reciprocity condition for (perceived) partitions, as well. To model the nature of network parti-

[1] This can easily be implemented with time-outs.

[2] For example, to detect failures FS_p could query a process, deeming it inaccessible if it does not repond in a timely fashion (inaccurate, but satisfying the requirement). We might put the onus on a process to announce its recovery.

tions, we require eventual reciprocity of inaccessibility suspicions. That is, if FS_p suspects q then eventually either FS_q suspects p or q fails.

A logical formula holds on a consistent cut. The membership of an *indexical set* of processes depends on when it is considered. In our model, 'when' translates to consistent cuts, the only physically-realizable instances. We use the following formulas and indexical sets to specify the behavior of FS_p.

- $\text{NotComm}_p(q)$holds on c if $q \in \mathsf{NotCommSet}_p(c)$
- $\text{Comm}_p(q)$ holds along c if $q \in \mathsf{CommSet}_p(c)$
- DOWN_q holds along $c = (h_1, \ldots, h_q, \ldots, h_n)$ if $crash_q$ is the last event in h_q
- UP_q holds along $c = (h_1, \ldots, h_q, \ldots, h_n)$ if $crash_q$ is not an event in h_q.

Non-triviality Conditions for FS_p

Crashes If q crashes, then eventually either p crashes or FS_p suspects q is unreachable:

$$\text{DOWN}_q \Rightarrow \Diamond\Big(\text{NotComm}_p(q) \vee \text{DOWN}_p\Big)$$

Recoveries If q begins executing and is reachable, then eventually either p crashes or FS_p suspects q is reachable:

$$\text{UP}_q \Rightarrow \Diamond\Big(\text{Comm}_p(q) \vee \text{DOWN}_p\Big)$$

Reciprocity If FS_p suspects q is inaccessible, then, if q does not crash, it eventually suspects p is inaccessible:

$$\text{NotComm}_p(q) \Rightarrow \Diamond\Big(\text{DOWN}_q \vee \text{NotComm}_q(p)\Big)$$

This is an artifact of p suspecting q: since p ceases communicating with q, p is, in fact, inaccessible to q.

Propagation Conditions for FS_p

Finally, we require failure suspectors to *gossip* among themselves.

Inaccessibility Propagation If FS_p believes, on cut c, it cannot communicate with q then it tries to propagate this belief to every FS_r for $r \in \mathsf{CommSet}_p(c)$:

$$\text{NotComm}_p(q) \Rightarrow \Diamond\Big(\text{NotComm}_r(q) \vee \text{NotComm}_r(p)\Big)$$

Accessibility Propagation If FS_p believes, along c, it can communicate with q then it tries to propagate this belief to every FS_r for $r \in \mathsf{CommSet}_p(c)$:

$$\text{Comm}_p(q) \Rightarrow \Diamond\Big(\text{Comm}_r(q) \vee \text{NotComm}_r(p)\Big)$$

3.1 Related work

Before discussing the other components, we discuss the relation between this and other work. In [7], Chandra and Toueg solve Distributed Consensus in an asynchronous system using a Failure Suspector, W, that satisfies certain (weak) requirements. [6] further shows that W is the weakest suspector that can be used to solve Distributed Consensus. While we do not consider consensus in this paper, we said in the Introduction that adding a majority requirement to the VSC abstraction, gives a simple, probabilistic solution to transaction commit. Since there are no fundamental differences between solving consensus and atomic commit problem, how are both approaches related (we will not, hereafter, distinguish consensus from atomic commit)?

First it should be clear that our Failure Suspector is not weaker than W. More important, [7] also places a majority requirement on processes before W can be used to solve consensus. To relate the two approaches, consider a generalization of consensus:

- suppose consensus is to be solved more than once, and let $consensus(i)$, for $i > 0$, be the i^{th} instance of the consensus problem;
- let Proc be the initial set of processes that solve $consensus(1)$;
- $consensus(i+1)$ begins only after $consensus(i)$ has been solved;
- for $consensus(i)$, $i > 1$, the processes chose their initial state randomly from the set $\{0, 1\}$.

In [7], $consensus(i)$ (for each i) would be solved by the same static set of processes Proc. The majority requirement to solve $consensus(i)$ is thus similar to a static voting scheme in the context of handling replicated data [11]. This is because [7] consider that failure suspicions are never stable: a process p believing $failed(q)$ can always change its mind.

In contrast, in the VSC model, failure beliefs are stable each time a new view is defined. Thus for $i \neq j$, $consensus(i)$ and $consensus(j)$ need not be solved by the same set of processes. Continuing the replicated data analogy, the majority requirement in the VSC model is similar to the dynamic voting scheme [9], which has been shown to lead to higher data availability than the static voting scheme.

4 The View Component

The view component operates whenever a link failure repairs, a process begins executing, recovers after a crash, and whenever the multicast component informs it that the current view has terminated (Section 5). $\textsc{vc}_p$ defines p's current view by interaction with other $\textsc{vc}$ components, and by using $\textsc{fs}_p$ information.

$\textsc{vc}_p$ defines a new view when it detects (or learns about through some other $\textsc{vc}$ component) agreement on $\mathsf{CommSet}_p()$ among the members of $\mathsf{CommSet}_p()$. The new view will be the largest subset of processes (containing p) satisfying this agreement.

4.1 The View Component Algorithm

In this section we outline how $\textsc{vc}_p$ detects or learns about $\mathsf{CommSet}_p()$ agreement.

When $\textsc{vc}_p$ is activated, it knows a near approximation of $\mathsf{CommSet}_p()$ from $\textsc{fs}_p$.[3] Whenever $\textsc{vc}_p$ receives an $\texttt{comm}(r)$ message from $\textsc{fs}_p$, it updates this approximation. Along cut c, $\textsc{vc}_p$ uses a deterministic function, $vc\text{-}Coord(p)$,[4] on the set $\mathsf{CommSet}_p(c)$ which returns a unique process identifier, and satisfies

$$\Big(\mathsf{CommSet}_p(c) = \mathsf{CommSet}_q(c)\Big) \Rightarrow \Big(vc\text{-}Coord(p) = vc\text{-}Coord(q)\Big).$$

For example, $vc\text{-}Coord(p)$ might be "choose the 'smallest' identifier from $\mathsf{CommSet}_p(c)$."

Each process also maintains a local counter, seq_p, which is incremented every time $\textsc{vc}_p$ considers $vc\text{-}Coord(p)$ to have changed (this is not necessarily every time $\mathsf{CommSet}_p(c)$ changes. For liveness, however, $vc\text{-}Coord(p)$ must change when $\textsc{vc}_p$ receives $\texttt{not-comm}(vc\text{-}Coord(p))$ from $\textsc{fs}_p$). The counter seq_p is initially zero and is essential in allowing us to define non-intersecting, concurrent views. The tuple (p, seq_p) fully describes p on any consistent cut.

Finally, the formula $\textsc{CommSetEq}(\mathsf{S})$ holds on c if and only if all $p \in \mathsf{S}$ have identical $\mathsf{CommSet}()$ sets at c. That is,

$$\textsc{CommSetEq}(\mathsf{S}) \stackrel{\text{def}}{=} \bigwedge_{p,q \in \mathsf{S}} \Big(\mathsf{CommSet}_p() = \mathsf{CommSet}_q()\Big)$$

In our protocol, each p sends its current $\mathsf{CommSet}_p()$ and current seq_p number to $vc\text{-}Coord(p)$ every time $\mathsf{CommSet}_p()$ changes.

4.2 Defining the New View

Let $\kappa = vc\text{-}Coord(p)$, and $\mathsf{S} = \mathsf{CommSet}_\kappa(c)$ for some cut c. Then $\textsc{vc}_\kappa$ receives $\mathsf{CommSet}_p()$ for $p \in \mathsf{S}$. Whenever it receives a different $\mathsf{CommSet}_p()$ from some p, $\textsc{vc}_\kappa$ discards the previous one and checks whether $\textsc{CommSetEq}(\mathsf{CommSet}_\kappa())$ holds. If it does, $\textsc{vc}_\kappa$ sets the new view, $\mathsf{View}_\kappa()$, to

$$\mathsf{View}_\kappa() = V = \Big\{(p, seq_p) \mid p \in \mathsf{CommSet}_\kappa()\Big\} \qquad (1)$$

The coordinator κ then sends the new view to each $\textsc{vc}_p$ (for $p \in V$) which then delivers the view to $\textsc{mc}_p$. $\textsc{mc}_p$ regains execution control and begins multicasting again. Unfortunately, as $\textsc{CommSetEq}(\mathsf{CommSet}_\kappa())$ is not a stable property (i.e. once true, forever true) we must take care in announcing the new view. We return to this issue in Section 6.

4.3 The Partial Order

Correctness of $\textsc{vc}_p$ means that the coordinator successfully sends the new view to the $\textsc{vc}$ components of all reachable members in the new view. We will henceforth use V to denote the (local) view that is agreed-upon by all the members of V.

[3] There may be notifications from $\textsc{fs}_p$ that have not yet reached $\textsc{vc}_p$.

[4] Technically, we should name some cut explicitly since the function's value depends p's indexical can-communicate-with set. We omit the cut reference, but with the understanding that $vc\text{-}Coord(p)$ has a temporal dependence. In fact p never knows which particular cut it is on, but at any point in its execution $\textsc{vc}_p$ has some set of process identifiers that satisfy a certain condition. It determines a coordinator by applying some rule to this set. The presence of c would only clarify matters for the omniscient reasoner.

Since process histories are linear, it makes sense to talk about the x^{th} version of a process's (local) view – we denote this by View_p^x.

Definition Given two agreement views V and V', $V \prec_I V'$ if and only if there is some p in $V \cap V'$ such that $V = \mathsf{View}_p^x$, and $V' = \mathsf{View}_p^{x+1}$. The transitive closure of $\prec_I$ is denoted $\prec$. □

It is not hard to see that the views defined by the collection of $\textsc{vc}_p$ components are partially ordered by $\prec$. We say V and V' are *concurrent* if and only if they are not $\prec$-related.

Proposition 4.1 trivially follows from the definition of views (Equation 1) and the increment rule for seq_p (all proofs can be found in [21]).

Proposition 4.1 *Let V and V' be concurrent views. Then $V \cap V' = \emptyset$.*

5 The Multicast Component

The Multicast Component of process p, $\textsc{mc}_p$, is responsible for implementing virtually-synchronous communication. $\textsc{mc}_p$ operates in two modes. In one mode it *multicasts* messages to the members of its current view $\mathsf{View}_p()$. In the other mode, it flushes outstanding multicasts to ensure they satisfy virtually-synchronous communication semantics, then *terminates* the current view. The transition from multicast mode to termination mode is triggered by any $\textsc{fs}_p$ `not-comm()` or `comm()` message. In this section, we define VSC semantics and the protocols $\textsc{mc}_p$ uses.

5.1 Definitions

Informally, virtually-synchronous communication is such that, for any view V, the processes of view V that mutually believe each other alive deliver the same set of multicasts.[5] To make the definition of VSC precise we need to define formally the set of messages considered to have been multicast in V, as well as the subset of processes that deliver them.

Definition Given a view V, message m is a *V-multicast* if it was sent by some p along a cut c such that $\mathsf{View}_p(c) = V$. □

[5] For simplicity, we omit other forms of communication. Non-multicast communications do not introduce new problems.

Definition (VSC) Let $V \prec_I V'$. Then communication in a system is *virtually-synchronous* if and only if all processes in V and in V' delivered the same set of V-multicasts. Moreover no message is delivered in more than one view. □

It is important to notice that process sequence numbers are not used in the definition. These are low-level pieces of information; the application layer should only be concerned with process identifiers. For an application-layer process, VSC ensures two processes that if they progress together from one view to another, then they delivered the same set of messages in the first view. As a result, if process state is determined by an initial state and the set of multicasts delivered to the process, VSC means that if processes begin executing in view V in the same state, then switch together to view V', they will begin executing in V' in the same state.

5.2 Two Modes of Operation

The component $\textsc{mc}_p$ operates in two modes:

1. in *normal* mode $\textsc{mc}_p$ reliably multicasts messages issued by the application layer of p, and delivers to the application layer multicasts it receives from other $\textsc{mc}$s;
2. in *view-termination* mode $\textsc{mc}_p$ does not multicast new messages; instead it attempts to flush outstanding multicasts to ensure the VSC semantics.

After receiving a view from $\textsc{vc}_p$, $\textsc{mc}_p$ is in normal mode. It enters view-termination mode as soon as it receives any (in)accessibility notification from $\textsc{fs}_p$. When view-termination mode ends, $\textsc{mc}_p$ gives control back to $\textsc{vc}_p$. $\textsc{mc}_p$ is inactive until it receives a new view from $\textsc{vc}_p$, whereupon $\textsc{mc}_p$ begins normal mode again.

5.3 $\textsc{mc}_p$ Normal Mode

Suppose $\textsc{vc}_p$ defines a view $V = \mathsf{View}_p()$ and delivers this to $\textsc{mc}_p$. Recall that views are sets of tuples, which we call process *signatures*:

$$\mathsf{View}_p() = \Big\{\sigma_q = (q, seq_q)\Big\}.$$

Upon receiving $\mathsf{View}_p()$, $\textsc{mc}_p$ enters normal mode, in which it multicasts and delivers messages. Each message m issued by the application layer of process p

is multicast by $\textsc{mc}_p$ to all $q \in V$. Before issuing the message, $\textsc{mc}_p$ adds σ_p to m. Let $sender(m)$ be the signature of the process from which m originated.

When $\textsc{mc}_p$ receives a message the following sequence of events occurs:

1. $\textsc{mc}_p$ delivers m (to the application layer) if $sender(m) \in V$, and discards m otherwise;
2. $\textsc{mc}_p$ also buffers any message it receives and delivers in V until it knows all other processes in V have received m.[6] When m is received by all processes in V we say it is *stable*.

By delivering only V-multicasts, the normal mode ensures that no multicast can be delivered in more than one view (see the VSC definition).

5.4 $\textsc{mc}_p$ View-Termination Mode

Consider a view $V = \mathsf{View}_p()$. Component $\textsc{mc}_p$ switches from normal mode to view-termination mode after receiving from $\textsc{fs}_p$ either **1)** `not-comm`(q) for $q \in \mathsf{View}_p()$, or **2)** `comm`(r) for $r \notin \mathsf{View}_p()$. This is because whenever a change in the communication topology is detected a new view must be defined reflecting that change. However, before defining a new view, $\textsc{mc}$ in view-termination mode must ensure the VSC definition is satisfied.

Once $\textsc{mc}_p$ enters view-termination mode, it need only consider relevant `not-comm`() events from $\textsc{fs}_p$ to terminate V. Thus, while executing in view-termination mode, $\textsc{mc}_p$ builds its own approximation of $\mathsf{NotCommSet}_p()$. This means failure notifications have a permanent effect until view-termination mode ends: `comm`(q) received by $\textsc{mc}_p$ in view-termination mode after `not-comm`(q) (for example due to a partition) cannot *undo* the `not-comm`(q) information.

Just as a new view for p is defined according to agreement on $\mathsf{CommSet}()$s, successfully terminating V involves partitioning V according to $\mathsf{NotCommSet}()$ agreement.

Definition The indexical set $\mathsf{Survives}_p(V)$ is V minus the set of processes $\textsc{mc}_p$ believes failed in V:

$$\mathsf{Survives}_p(V) = V - \Big\{(q, seq_q) \mid \textsc{NotComm}_p(q)\Big\}$$

□

[6] There are many standard ways of achieving this – e.g. piggybacking information on messages.

Before we can explain how to ensure VSC, we need the following data structures.

Definition Consider $V = \mathsf{View}_p()$ and consistent cut c. The vector $msg_p(V, c)$ (of size $\mid V \mid$) is defined such that:

- its p^{th} component, $msg_p(V, c)[p]$, is the number of V-multicasts that originated from p (up to c);
- for $q \in V$, $q \neq p$, its q^{th} component, $msg_p(V, c)[q]$, is the number of V-multicasts $\textsc{mc}_p$ delivered up to c that originated from q. □

Definition (View Terminated) Consider view V and S such that $\emptyset \neq \mathsf{S} \subseteq Ids(V)$ (where $Ids(V)$ is the set of process identifiers appearing in V). Then $\textsc{vt}(V, \mathsf{S})$ holds along cut c if and only if

$$\bigwedge_{p,q \in \mathsf{S}} \Big(\Big(msg_p(V, c) = msg_q(V, c)\Big) \wedge \Big(\mathsf{Survives}_p(V, c) = \mathsf{Survives}_q(V, c)\Big)\Big)$$

It is not hard to see $\mathsf{S} = Ids(\mathsf{Survives}_p(V))$. □

In other words $\textsc{vt}(V, \mathsf{S})$ is true exactly when the processes in S agree on both the messages multicast in V and on their respective $\mathsf{Survives}(V)$ sets. For $\textsc{mc}_p$, detecting termination of $V = \mathsf{View}_p()$ is thus reduced to detecting $\textsc{vt}(V, \mathsf{S})$ (for $p \in \mathsf{S} \subseteq Ids(V)$).

Having detected $\textsc{vt}(V, \mathsf{S})$, whether $\mathsf{S} = Ids(V)$ or $\mathsf{S} \subset Ids(V)$ is important in determining the new view. In the first case, whatever view, V', $\textsc{vc}_p$ later defines, VSC is satisfied with respect to the pair (V, V'). In the second case $\textsc{mc}_p$ must pass $\mathsf{Survives}_p(V)$ to $\textsc{vc}_p$; we will want the new view to be a subset of $\mathsf{Survives}_p(V)$.

To guarantee that every non-crashed process in V eventually detects $\textsc{vt}(V, \mathsf{S})$ for some S, $\textsc{mc}_p$ behaves as follows in view-termination mode:

- it stops multicasting new messages;[7]
- it rejects any message m such that $sender(m) \notin \mathsf{Survives}_p(V)$.
- upon receiving `not-comm`(q) from $\textsc{mc}_p$ (for $q \in V$), $\textsc{mc}_p$ signs and forwards any V-multicasts originating from q that are still in p's buffer (Section 5.3). $\textsc{mc}_p$ then removes these messages from

[7] If the network were a broadcast domain, $\textsc{mc}_p$ could continue multicasting using a new signature $(p, seq_p + 1)$. The problem for less general environments is that the new multicast view (destination set) is not yet known.

its buffer. MC_q rejects the re-issued message if $\text{NOTCOMM}_q(p)$ holds (i.e. if MC_q has received `not-comm`(p) from FS_q).[8]

Proposition 5.1 *Consider view-termination mode as described above. Then for each $p \in V$, there exists a set, S_p such that $p \in \mathsf{S}_p$ and $\text{VT}(V, \mathsf{S}_p)$ holds.*

5.5 An Algorithm to Detect $\text{VT}(V, \mathsf{S})$

Like the VC_p algorithm detecting $\text{COMMSETEQ}()$, the MC_p algorithm detecting $\text{VT}(V, \mathsf{S}_p)$ relies on a coordinator process. MC_p determines its view-termination coordinator with a deterministic function, $mc\text{-}Coord(p)$, on the set $\mathsf{Survives}_p(V, c)$. We require that for p and q in V, with identical $\mathsf{Survives}(V)$ sets, $mc\text{-}Coord(p) = mc\text{-}Coord(q)$.

Let $\chi = mc\text{-}Coord(p)$. Then χ attempts to detect $\text{VT}(V, \mathsf{Survives}_\chi(V))$. MC_p also increments the sequence number counter, seq_p, whenever MC_p considers $mc\text{-}Coord(p)$ to have changed (for liveness, the function $mc\text{-}Coord(p)$ must change whenever MC_p receives `not-comm`$(mc\text{-}Coord(p))$ from FS_p).

Process p sends $msg_p(V)$, $\mathsf{Survives}_p(V)$, and seq_p to $mc\text{-}Coord(p)$ when MC_p first considers $mc\text{-}Coord(p)$ to be its coordinator, and whenever $msg_p(V)$ and $\mathsf{Survives}_p(V)$ are modified. If $\chi = mc\text{-}Coord(p)$, then:

$$\text{VT}(V, \mathsf{Survives}(V)) \Leftrightarrow \bigwedge_{p \in \mathsf{Survives}_\chi(V)} \Big(msg_\chi(V) = msg_p(V) \wedge \mathsf{Survives}_\chi(V) = \mathsf{Survives}_p(V) \Big)$$

Proposition 5.2 *Consider a view V, with $p \in V$ and the view-termination protocol described above. Then eventually, either p crashes or it detects $\text{VT}(V, \mathsf{Survives}_\chi(V))$.*

Finally, the fact that $\text{VT}(V, \mathsf{S})$ is not stable poses the same problems as those posed by $\text{COMMSETEQ}()$'s instability. We consider both in the next section.

6 Instability of $\text{COMMSETEQ}()$ and $\text{VT}(V, \mathsf{S})$

As described in the previous sections, once VC_p learns $\text{COMMSETEQ}(\mathsf{CommSet}_p())$ it switches control to MC_p; switching control from MC_p to VC_p is based on detecting $\text{VT}(\mathsf{View}_p(), \mathsf{S})$. In both cases, the relevant property is not stable – it may become false after holding along some cut. Let `switch`(VC, V') be the message announcing the new view, V', and `switch`$(\text{MC}, \mathsf{Survives}())$ be the message announcing termination of view V.

Since neither $\text{COMMSETEQ}(\mathsf{S})$ nor $\text{VT}(V, \mathsf{S})$ are stable properties, we can arrive at the following situation[9]:

- Take $p, q \in V$ such that p and q believe each other accessible, and let κ be their mutual VC coordinator ($\kappa = vc\text{-}Coord(p) = vc\text{-}Coord(q)$). Suppose VC_κ determines the new view, V' ($\kappa, p, q \in V'$), sends `switch`(VC, V') to p only, and then crashes. VC_p, upon receiving `switch`(VC, V'), adopts $\mathsf{View}_p() = V'$ and switches control to MC_p in normal mode.

- Now suppose that in addition to VC_q *not* getting `switch`(VC, V'), FS_q notifies VC_q that κ is inaccessible; q continues executing in VC_q waiting for some new coordinator κ' to inform it of the new view. In particular, suppose $\kappa' = p$.

- Since p and q continue to believe each other accessible, FS_q gossips `not-comm`(κ) to FS_p. At this point, MC_p enters view-termination mode for view $\mathsf{View}_p() = V'$, and q is still executing in VC_q waiting to receive the successor view to V. Observe that unless one of the processes crashes or a network partition splits them, p and q need never believe each other inaccessible.

- For VC_q to make progress, its coordinator VC_p must tell it some new view. Unfortunately, VC_p cannot begin executing until MC_p leaves view-termination mode. MC_p cannot leave view-termination mode until it receives $\mathsf{Survives}_q()$ from MC_q (after all, $q \in V'$ and $q \in \mathsf{CommSet}_p()$). In other words, p and q are *deadlocked* because their execution controls are out of phase. The control discrepancy prevents either one (VC_q or MC_p) from making progress until one of them believes the other inaccessible – q is stuck in VC_q, and p is stuck in MC_p.

While processes being out of phase is not always destructive, and in fact is quite natural whenever partitions occur, it is destructive in this case since it induces deadlock. The following precludes deadlock.

[8] Duplicate messages are recognized and discarded as usual.

[9] While we illustrate instability with $\text{COMMSETEQ}()$ and the switch from VC_p to MC_p, a similar situation arises for $\text{VT}(V, S)$ as well.

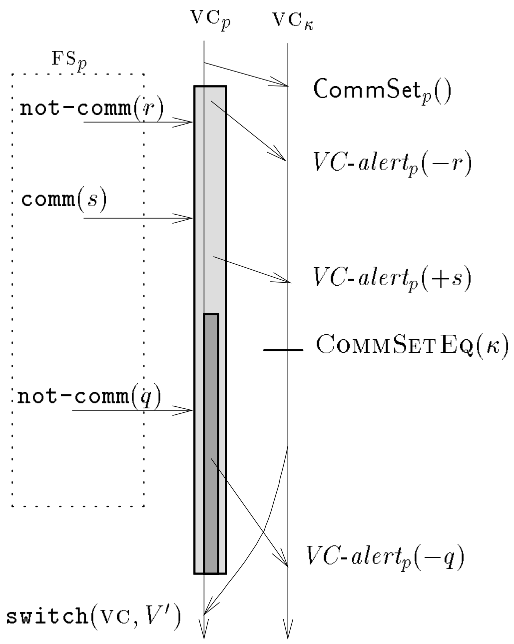

Figure 2: $FS\text{-}VC\text{-}Notify_p(v')$ (lightly-shaded rectangle), $VC\text{-}alert_p()$, $FS\text{-}VC\text{-}Late_p$ (darkly-shaded rectangle)

6.1 Component-Switch Protocol

Let κ be shorthand for *vc-Coord(p)* when VC_p is executing. We describe the protocol only for the switch from VC_p to MC_p; the situation is analogous for the reverse switch. Let $V = \mathsf{View}_p()$. We define the following concepts as depicted in Figure 2:

- From Section 4, each accessibility notification from FS_p forces VC_p to inform its coordinator VC_κ of the change to $\mathsf{CommSet}_p()$. Let $VC\text{-}alert_p()$ denote the message VC_p sends to VC_κ to inform VC_κ of the change to $\mathsf{CommSet}_p()$.
- Let $FS\text{-}VC\text{-}Notify_p(V')$ be the set of $\texttt{not-comm}(q)$ and $\texttt{comm}(r)$ accessibility notifications VC_p received from FS_p after sending its first $\mathsf{CommSet}_p()$ to *any* coordinator and before receiving $\texttt{switch}(\text{VC}, V')$ from VC_κ;

So given V' and $FS\text{-}VC\text{-}Notify_p(V')$, VC_p can infer which $VC\text{-}alert_p()$ messages reached VC_κ before it detected $\text{COMMSETEQ}(\mathsf{CommSet}_\kappa())$ and which did not. Let $FS\text{-}VC\text{-}Late_p$ be the subset of $FS\text{-}VC\text{-}Notify_p(V')$ for which the corresponding $VC\text{-}alert_p()$ message did not reach VC_κ.

The *Component Switch* protocol for VC_p is:

1. The coordinator κ sends the $\texttt{switch}(\text{VC}, V')$ message using a *best effort reliable multicast* [13] (a process receiving the message reissues it to all the destination processes).
2. Upon receiving $\texttt{switch}(\text{VC}, V')$, VC_p:
   (a) *logically reorders* it to be *before* VC_p sent any of the messages in $FS\text{-}VC\text{-}Late_p$ (this will be clearer after 3) ;
   (b) installs V' as $\mathsf{View}_p()$ and switches control to MC_p, in normal mode;
3. MC_p handles messages in $FS\text{-}VC\text{-}Late_p$ as if the corresponding notifications from FS_p had just arrived (i.e. while MC_p is executing, and not while VC_p was executing). Specifically, MC_p simulates receiving these accessibility notifications in $\mathsf{View}_p() = V'$.

Proposition 6.1 *The Component-Switch Protocol prevents deadlock.*

7 Concluding Remarks

This paper has shown how to implement virtually-synchronous communication using a three-component architecture for systems that experiences process crash failures and network partitions. The three-component architecture lead us to define a clear semantics for a Failure Suspector (a necessary part of any live, asynchronous system) that guarantees liveness of the VC and MC components. Clearly defining these semantics allows one to implement the Failure Suspector as a modular tool – distinct from all other components – whose implementation can take advantage of the characteristics of the underlying network.

Considering a membership service in relation to virtually-synchronous communication also lead us to better understand the need for a *strong-partial* compared to a *weak-partial* membership service. Specifically, a strong-partial membership service (non-intersecting concurrent views) is naturally related to virtually-synchronous communication. We can understand this in the following way. The MC component must identify the sender of a message by its signature σ_q to ensure that no multicast is delivered in more than one view. This led us to define a view as a set of process signatures. Considering the increment conditions of seq_p, two different views V and V' trivially have a non-empty intersection. In other words, by requiring that no multicast be delivered in more than one view, we were led to the partial-strong membership service. However if we remove the MC component, (i.e. if the membership service is only defined by FS and VC, without any reference to communication), then the sequence number seq_p has no clear justification. In that case, a view is just a set of process identifiers (or a set of identifiers and an incarnation number). With this definition, the same VC protocol we described would define concurrent views that overlap, providing only a weak-partial membership service.

References

[1] A. El Abbadi, D. Skeen, and F. Cristian. An Efficient, Fault-Tolerant Algorithm for Replicated Data Management. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 215–229. A.C.M., 1985.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms in Broadcast Domains. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth WDAG; Israel*, pages 292–312. Springer-Verlag, 1992. LNCS 647.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 123–138, November 1987.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[5] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report TR-91-1216, Cornell University, July 1991.

[6] T. D. Chandra and V. Hadzilacos andS. Toueg. T Weakest Failure Detector for Solving Consensus. In *Proceedings of the 11th Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 147–158. ACM, August 1992.

[7] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proceedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 325–340. ACM, August 1991.

[8] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *A.C.M. Transactions on Computer Systems*, 3(1):63–75, 1985.

[9] D. Davcev and W. A. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 87–96, 1985.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[11] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating System Principles*, pages 150–159, December 1979.

[12] F. Jahanian and W. M. Moran. Strong, Weak and Hybrid Group Membership. In *Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data*, pages 34–38, November 1992.

[13] T. Joseph and K. Birman. *Distributed Systems*, chapter Reliable Broadcast Protocols, pages 293–317. Addison-Wesley, 1989.

[14] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the A.C.M.*, 21(7):558–565, 1978.

[15] K. Marzullo and G. Neiger. Detection of Global State Predicates. In *Proceedings fo the Fifth International WDAG*, pages 254–272. Springer-Verlag (LNCS 579), 1991. Delphi, Greece.

[16] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Proceedings of the IEEE 11th ICDCS*, pages 480–488, May 1991.

[17] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. In *Proceedings of the IEEE International Working Conf on Dependable Computing for Critical Applications*, pages 137–145, February 1991.

[18] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Procedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 341–351. A.C.M., August 1991.

[19] A. M. Ricciardi, K. P. Birman, and P. Stephenson. The Cost of Order in Asynchronous Systems. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth WDAG; Israel*, pages 341–352. Springer-Verlag, 1992. LNCS 647.

[20] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proceedings of the IEEE 13th ICDCS*, May 1993.

[21] S. Schiper and A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. Technical Report TR93-1339, Cornell University, 1993.