Impossibility of (Repeated) Reliable Broadcast

Aleta Ricciardi

TR TR-PDS-1996-003 April 1996

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Impossibility of (Repeated) Reliable Broadcast (revised of TR-PDS-1995-016)

Aleta Ricciardi*

April 6, 1996

Abstract

We consider the problem of *Repeated Reliable Broadcast (RRB)* in asynchronous systems, and show that in the absence of infinite storage capacity, solutions to RRB require *perfect failure detectors*. In the process, we show that the common reason used to explain the impossibility of Distributed Consensus is incorrectly attributed: it is not the inability of a process to know whether a remote process is crashed that makes Consensus impossible. We then generalize the results for RRB to a broad class of problems, which we call Generic Repeated Coordination. We consider some of the implications of our results for protocols that assume reliable broadcast, and show that the need for reliable channels distinguishes Atomic Broadcast from Consensus, which even under weaker channel assumptions, can still be solved with $\diamond\mathcal{W}$ failure detectors. In the absence of perfect failure detectors and of processes that never crash, this work illuminates the need for a more flexible notion of "non-faulty process".

*Supported by the École Polytechnique Fédérale de Lausanne and a University of Texas Research Institute Summer Research Fellowship.

1 Introduction

Coordination and agreement problems in distributed systems have been researched extensively (Hadzilacos and Toueg have compiled a large survey [11]), but the overwhelming body of literature discusses the issues in reaching a single distributed decision, or when multiple decisions must be reached, the issues in determining their order. While many primitive agreement constructs are used continually in distributed databases (*e.g.*, atomic commit, concurrency control [1]) and distributed services (*e.g.*, reliable broadcast [3], leader election), the issues in ensuring coordination can occur infinitely often have not been examined. Here we are concerned with solving distributed coordination problems repeatedly given the reality of finite storage space and the need to maintain information pertaining to decisions for all correct processes (*i.e.*, processes that never crash).

We consider first *Repeated Reliable Broadcast (RRB)*, and show that solutions require either *perfect failure detectors* [5] or infinite storage capacity. While falling media prices may make it seem that storage space is, for all intents and purposes 'infinite', sliding window protocols, which are the typical implementations of reliable channels, accommodate on the order of ten distinct messages [9]. These results point to subtle, yet significant, assumptions made by any protocol using RRB. Inspecting these protocols, one finds reliable channels are used as infinite, fault-tolerant storage space. Given that RRB is so widely assumed, we give conditions under which *some-chance channels* (communication channels with the property that each message has non-zero probability of reaching its destination) suffice. We then discuss the implications with respect to implementing strong failure detectors from weak ones, for solving Consensus, and for solving Atomic Broadcast [5]. We show that some-chance channels suffice in the first two cases, but not the last. That is, when storage space considered in the computing model, Consensus is *weaker* than both RRB and Atomic Broadcast, because, in the absence of reliable channels, it can still be solved with only weak failure detectors.

We extend the results for RRB to a broad class of problems, which we define as *Generic Repeated Coordination*. In Generic Repeated Coordination, processes in some known set, S, are repeatedly making *decisions* together. The decision itself can represent "deliver message m in position i", "the new leader is process p", "add the site s as a service provider", "increment local copies of x by four", or "take a checkpoint". We are not concerned with how the members of S actually arrive at the decision as this has been treated in many places. For example, [2, 8, 15, 11] all discuss atomic multicast, [1, 16, 21] all treat atomic commit, and [6, 14, 13] cover check-pointing. Our focus is on ensuring infinitely many decisions *can* be made, not on how any single decision is reached.

An interesting corollary of our results shows that the commonly-stated reason behind the impossibility of Distributed Consensus, the inability to know whether a remote process is crashed, is incorrectly attributed. While unreliable failures detectors suffice for Distributed Consensus, our results show neither they nor eventually reliable failure detectors provide this knowledge.

Section 2 presents the asynchronous computing model and some important system properties. Section 3 defines Repeated Reliable Broadcast and Section 4 discusses the special conditions imposed by having to maintain decision information for all correct processes. In Section 5 we prove perfect failure detectors are needed for both RRB and the broader problem of Generic Repeated Coordination. Section 6 shows that the need for reliable channels distinguishes Atomic Broadcast from Consensus and discusses other ramifications of our results; we conclude in Section 7.

2 The System Model and Definitions

The system consists of a (finite, for these purposes) set, S, of processes that communicate by passing messages over a network of channels (Section 2.1). The system is asynchronous in that neither processes' relative speeds nor message transmission delays have fixed bounds. Processes fail by crashing (permanently) but otherwise follow their assigned protocol.

Throughout this paper, we assert various System Properties to highlight them especially. While these assumptions are all common in the literature and not controversial, we state them here to avoid confusion and to be able to refer to them precisely. When relevant, we formalize them after we have developed the formal language and storage model.

System Property 1 *Process failures are unpredictable; they are not part of any protocol a process executes, or hardwired into its state or code. Consequently, if a process is not currently crashed, it has some chance of never crashing.*

Each process has local state that changes whenever the processes executes an *event*; of particular interest are the events $crash_p$ (used to model p crashing), $send_p(q, m)$ (p sends m to q on the channel connecting them) and $recv_q(p, m)$ (q receives m from the channel connecting it to p). It is important to note that $send(,)$ and $recv(,)$ are low-level actions, distinct from the high-level 'broadcast' and 'deliver' actions we will define for Repeated Reliable Broadcast. After each event, a process makes all logical inferences from its new local state. This includes, for message delivery, reading the message and inferring the consequences of the message contents. A history for process p, h_p, is a sequence of events beginning with the event $start_p$. Common assumptions about the structure of histories (*e.g.*, the *start* event occurs only once) are omitted. A *cut* over S is a tuple of finite process histories, one for each $p \in \mathsf{S}$. The unique cut c_0 consists of only the $start_p$ events for each $p \in \mathsf{S}$. Following Lamport [15], we define the relation $e \xrightarrow{1}_c e'$ for a cut $c = (h_1, \ldots, h_n)$ (e and e' are immediately causally related in c) if either e is the i^{th} event of h_p and e' is the $(i+1)^{th}$ event of h_p, or if $e = send_p(q, m)$ in h_p and $e' = recv_q(p, m)$ in h_q. Inter-event causality (written $e \rightarrow_c e$) is the irreflexive, transitive closure of the $\xrightarrow{1}_c$ relation.

As inter-event causality reflects progress made between events, the causal relation between two cuts reflects progress made between global states. Formally, given c_1 over $\mathsf{S}_1 \subseteq \mathsf{S}$ and c_2 over $\mathsf{S}_2 \subseteq \mathsf{S}$, define $c_1 \leq c_2$ if and only if $\mathsf{S}_1 \cap \mathsf{S}_2 \neq \emptyset$ and, for any process in the intersection, its history component in c_1 is a prefix of its history component of in c_2. Two cuts c_1 and c_2 are *indistinguishable* to process p (written $c_1 \sim_p c_2$) when p's history component (state) in both is identical; given $\mathsf{S}_1 \subseteq \mathsf{S}$, then $c_1 \sim_{\mathsf{S}_1} c_2$ means $c_1 \sim_p c_2$ for each $p \in \mathsf{S}_1$.

A cut c is *consistent* if given any $c \leq c_1$ such that $e \rightarrow_{c_1} e_1$ and $e_1 \in c$, then $e \in c$ [6]. A *system run* is a cut over S in which all histories are infinite.[1]

System Property 2 *A process can only store (state) information on physical media of one form or another, and physical storage space is finite.*

System Property 3 *When a process crashes, any stored state, though non-volatile, is unavailable to others. As well, no process can store information at, or retrieve information from, a remote process, except through that process.*

2.1 Communication Channels

The following definitions clarify the model's communication network. As, ultimately, our results prove statements about reliable channels, we cannot assume them in our model.

Definition 1 *A communication channel is* some-chance *if it guarantees that each message sent through it has some non-zero probability of reaching its destination.*

Definition 2 *A communication channel is* reliable *if it guarantees that any message sent through it will eventually be received provided both the sender and receiver do not crash.*

In our model, all channels between processes are some-chance channels; reliable channels, the abstraction implemented by solutions to RRB, are built from some-chance channels.

Proposition 1 *Suppose processes p and q are correct and connected by a some-chance channel. If p sends an infinite number of messages to q, then q receives an infinite number of them.*

Proof Let $m_1, m_2, \ldots$ be an infinite sequence of messages sent by p to q, and let $0 < \epsilon < 1$ be the probability that m_i is not lost by the channel. Define $X_1, X_2, \ldots$ to be independent and identically distributed random variables, representing the successful transmission of message m_i; that is $Pr(X_i = 1) = \epsilon$. Given prefix h_p, $|h_p|$ is the number of events in h_p, and define $N(h_p)$ to be the number of messages sent to q in h_p. From our assumptions, $N(h_p) \rightarrow \infty$ (only) as $|h_p| \rightarrow \infty$. Define $N^\epsilon(h_p)$ to be the number of the $N(h_p)$ messages sent that will be successfully transmitted,

$$N^\epsilon(h_p) \stackrel{\text{def}}{=} \sum_{i=1}^{i=N(h_p)} X_i \; .$$

[1] Histories of crashed processes have infinitely many trailing *crash* events.

We claim that $N^{\epsilon}(h_p) \to \infty$ as $|h_p| \to \infty$. To see this, note that

$$\lim_{N(h_p)\to\infty} \left(\frac{1}{N(h_p)} \sum_{i=1}^{i=N(h_p)} X_i \right) \to \epsilon \text{ almost surely, and therefore,}$$

$$\lim_{N(h_p)\to\infty} N^{\epsilon}(h_p) = \lim_{N(h_p)\to\infty} \Big(N(h_p) \times \epsilon \Big) \to \infty \text{ almost surely.}$$

The correspondence between $N(h_p)$ and $|h_p|$ means that $N^{\epsilon}(h_p) \to \infty$ (only) as $|h_p| \to \infty$. $\boxed{\text{QED}}$

In consequence, if p sends message m infinitely often to q, eventually q will receive m.

2.2 The Formal Language

We use a logic supporting temporal and knowledge modalities to define the problem. Formulas are evaluated on consistent cuts in the context of a run.[2] When formula ϕ holds on cut c of run r, we write $(r, c) \models \phi$. The modalities have the following semantics:

Time $(r, c) \models \Box\phi$ (always) if and only if $(r, c) \models \phi$, and $\forall c' \in r, c \leq c' : (r, c') \models \phi$.

(read: for all c' in r such that $c \leq c'$, (r, c') satisfies ϕ)

$(r, c) \models \Diamond\phi$ (eventually) if and only if $\exists c' \in r, c \leq c' : (r, c') \models \phi$.

$(r, c) \models \ominus\!\!\!\!\Diamond\phi$ (in the past) if and only if $\exists c' \in r, c' \leq c : (r, c') \models \phi$.

$(r, c) \models \boxminus\phi$ (always in the past) if and only if $\forall c' \in r, c' \leq c : (r, c') \models \phi$.

Knowledge From Halpern and Moses[12],

$(r, c) \models K_p\phi$ (p knows) if and only if $\forall c' \sim_p c : \forall r', c' \in r' : (r', c') \models \phi$.

$(r, c) \models D_{\mathsf{S}}\phi$ (distributed knowledge within S) if and only if

$\forall c' \sim_{\mathsf{S}} c : \forall r', c' \in r' : (r', c') \models \phi$.

The semantics of the tense modalities involve only the run in which the formula is being evaluated while the semantics of the knowledge modalities quantify over all runs. Knowledge is intended to express a process's understanding of its surroundings, and the semantics of knowledge modalities account for global state from the process's perspective. At any point in an ongoing run (that is, at any consistent cut), a process cannot determine which particular run will unfold, or even which global state, among those it can imagine are plausible, is the actual one. We stipulate that crashed processes cannot 'know' anything as it is incapable of reasoning about its environment; that is, $K_p\phi \Rightarrow \neg\text{CRASH}(p)$.

[2]This is equivalent to the strict linear descriptive model given in [12] in which a cut is a linear sequence of events that respects causality.

From System Property 1, if a process is *inevitably* going to crash, it already is crashed,

$$\forall c : \forall r', c \in r' : (r', c) \models \Diamond\text{CRASH}(p) \text{ then } (r', c) \models \text{CRASH}(p).$$

A formula expressing knowledge or only the past or present that holds on a pair (r, c) necessarily holds on all pairs (r', c) for which $c \in r'$. In such instances we write $c \models \phi$, rather than restrict to a specific run.

Throughout, propositions expressing the fact that process p has executed some event, such as $\text{RECV}_p(q, m)$ or $\text{CRASH}(p)$, are defined to be stable (once true, always true). For example, $(r, c) \models \text{RECV}_p(q, m)$ if and only if p's history component in c contains the event $recv_p(q, m)$.

3 Repeated Reliable Broadcast

Repeated Reliable Broadcast (RRB) begins with the simple definition for Reliable Broadcast [3, 5], and adds Unboundedness and Transience clauses to make the repeated nature of the problem explicit. Validity, Integrity and Agreement are as in [5]. The difference between Reliable Broadcast and Uniform Reliable Broadcast [10, 20] is that in Uniform Reliable Broadcast a process cannot deliver a message until it knows the message will persist for correct processes despite other processes' failures, the passage of time, and network anomalies. In Reliable Broadcast, a process can deliver a message then try to propagate it to other processes, knowing that safety will not be violated if it crashes.

Below, formulas expressing the clauses of RRB must hold on all cuts of all runs generated by every solution.

Definition 3 (Repeated Reliable Broadcast) *Given a set of processes,* S*, let* M *be an infinite set of all possible messages that processes deliver with the special, local event* $\text{deliver}_p(m)$ *for* $p \in \mathsf{S}, m \in M$*:*[3]

Unpredictability *Messages to be delivered are broadcast by members of* S *but (Input) external factors dictate which messages are generated in a run and (Location) the member of* S *at which a message originates is not known in advance.*

Transience *External factors responsible for generating a message may not recur.*

Agreement *If any correct process* p *in* S *executes the event* $\text{deliver}_p(m)$*, then every other correct process* q *in* S *eventually executes* $\text{deliver}_q(m)$*,*

$$p \in \mathsf{S} \wedge \Box\neg\text{CRASH}(p) \wedge \text{DELIVER}_p(m) \Rightarrow \bigwedge_{q \in \mathsf{S}} \Diamond\Big(\text{DELIVER}_q(m) \vee \text{CRASH}(q)\Big). \qquad (1)$$

[3]The events $deliver_p(m)$ and $recv_p(q, m)$ are different. The latter is a low-level primitive event whereby p takes the message m from the some-chance channel between q and itself; the former is the higher-level event p executes in whatever protocol it is running that implements reliable channels on top of some-chance channels.

Integrity *The obligation to execute* $\mathrm{deliver}_p(m)$ *only applies if* m *is broadcast by a member of* S.

Validity *A correct process delivers all messages it broadcasts.*

Unboundedness *In any run, the number of messages that will be broadcast by a correct process is not known or fixed in advance.*

Unpredictability means the messages to be broadcast in a run are not predetermined. In Distributed Consensus, a similar clause means the initial condition values are randomly set; in RRB these unpredictable initial conditions (may) exist for each message. For message, m, let o_m denote the process at which m originated.[4] Transience can equally be stated as, once distributed knowledge of a message is lost within S, it may never (is not guaranteed to) be re-attained; it puts the onus on processes in S to store messages for all correct processes (the external world is not guaranteed to regenerate the conditions that caused any given message to be broadcast). Integrity prevents processes from simply delivering every message in the set M (in Atomic Commit, a similar clause prevents processes from always deciding "abort"). Agreement is directly from [5]. Validity precludes solutions in which messages are never delivered (and System Property 1 precludes solutions in which all processes purposely crash themselves). Unboundedness captures the nature of applications and systems that use Reliable Broadcast.

3.1 Solutions to RRB

We use Φ_m to denote the logical formula representing the fact that the message m has arisen in a run to be broadcast and delivered reliably; we will also (ab)use Φ_m to denote the logical formula representing the contents of the message m. Integrity means that a process that has never known a message has arisen, cannot have delivered it

$$\boxminus \neg K_p \Phi_m \Rightarrow \neg \mathrm{DELIVER}_p(m) \ . \tag{2}$$

A solution to RRB that tolerates f process crashes makes sure (the contents of) every message that has been delivered by a correct process persists and is available to all correct processes despite the failure of f members of S. Given S, let $\mathsf{nc}(\mathsf{S}, c)$ be the non-crashed processes at cut c,[5]

$$\mathsf{nc}(\mathsf{S}, c) \ \stackrel{\mathrm{def}}{=} \ \left\{ p \in \mathsf{S} \mid c \models \neg \mathrm{CRASH}(p) \right\} \ .$$

As the composition of $\mathsf{nc}(\mathsf{S}, c)$ always depends on the cut on which it is evaluated, and as formulas are evaluated on consistent cuts, in temporal formulas involving $\mathsf{nc}(\mathsf{S}, c)$ we will take the cut determining the set's composition to be the cut on which the formula is being evaluated. For example, we will write $(r, c) \models \Diamond D_{\mathsf{nc}(\mathsf{S})} \phi$ rather than $(r, c) \models \exists c' \in r, c \leq c' : D_{\mathsf{nc}(\mathsf{S}, c')} \phi$.

[4]Unpredictability means the identity of o_m depends on each run; in this notation, the run is not explicitly given.

[5]This set is defined and identical for any run containing c.

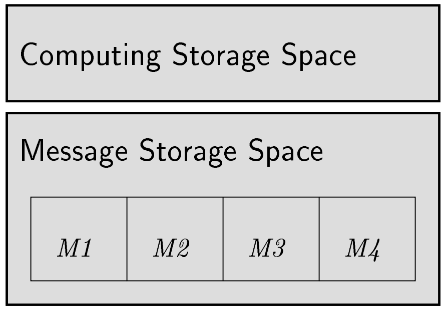

Figure 1: Illustration of a process's storage space.

Definition 4 *A formula ϕ is* crash-persistent in S*, written* C-PERSIST(S, ϕ)*, exactly when it will always be distributed knowledge among the non-crashed processes of* S*,*

$$\text{C-PERSIST}(\mathsf{S}, \phi) \stackrel{\text{def}}{=} \Box D_{\mathsf{nc(S)}} \phi \ .$$

The Transience property of RRB is the incentive for processes making messages that are to be reliably delivered crash-persistent. Transience implies that if a message is no longer distributed knowledge among the non-crashed processes, there are runs completing that cut throughout which it will never be learnable,

$$\forall m \in M : \forall c : c \models \diamondminus D_{\mathsf{nc(S)}} \Phi_m \ \wedge \ \neg D_{\mathsf{nc(S)}} \Phi_m \text{ then } \exists r', c \in r' : (r', c) \models \Box \neg D_{\mathsf{nc(S)}} \Phi_m \ . \qquad (3)$$

A crash-persistent message will always be learnable to correct processes of S. All solutions to RRB must ensure (the contents of) every message delivered by a correct process is crash-persistent. For RRB, this means every process can do one of the following:

1. Keep a copy of every message it delivers. If it is correct, these messages will always be available for other correct processes; if it is not correct, safety is not violated.

2. Not retain a copy of the messages it delivers, but (to tolerate f crashes) have $f + 1$ other processes store them. If the process is correct, at least one copy will survive.

Thus, the last component of our model is storage space, which we divide into a process's computational storage space, and its message storage space, denoted msg-store$_p$. For simplicity, we assume all messages are the same size, and that each process can buffer (store) B different messages. Figure 1 displays this layout for a process, with $B = 4$. Processes do not use their computational storage space to store messages.

3.2 The Storage Model

Neither of the above options is attractive for a theoretical discussion. The second option prevents a process from delivering a message until it is stored at $f+1$ other processes, and requires processes to store messages they do not deliver. In the first option, to assert C-PERSIST(S, Φ_m) is to imply that either $f+1$ different processes have delivered and stored Φ_m, or that one of the $g \leq f$ process that have delivered and stored Φ_m is correct. Most important however is that in the first option, once msg-store$_p$ fills up, p must make sure any message m it overwrites in msg-store$_p$ becomes crash-persistent on the chance that p itself is correct. That is, lacking the ability to correctly predict which processes (including itself) will never crash, p must eventually revert to the second option and replicate Φ_m at $f+1$ other processes (whose own msg-stores will eventually fill up). We therefore further separate message storage space so that given the B buffers of msg-store$_p$, each process reserves $l < B$ slots for local, non-crash-persistent storage, and 'donates' the remaining $B-l$ slots to implement S-wide crash-persistent storage. If $\delta = \lfloor n(B-l)/(f+1) \rfloor$, then δ different messages can be in crash-persistent storage simultaneously. We are hereafter concerned with these δ slots.

4 Storing and Purging Information

When a process is only exempt from delivering messages if it has crashed, every process that never crashes must have access to messages delivered by other correct processes, no matter how late in the computation and no matter how many other processes have crashed. Suppose all δ slots in crash-persistent space are filled, and that all $n \times l$ local non-crash-persistent slots are filled; consider any Φ_m stored (whether in crash-persistent storage or not). If q should deliver this message, but is not yet known to have done so, there is a dilemma:

- Keeping Φ_m stored prevents any process from delivering a $(\delta + l + 1)^{th}$ message. Without space to move a locally stored message to communally replicated crash-persistent storage, we have a liveness violation.
- On the other hand, if Φ_m is overwritten, Transience means Φ_m may never be learnable within S. In this case, unless q can be guaranteed to crash, Agreement could be violated.

Consequently we must understand the conditions under which it is safe to remove messages from crash-persistent storage so that processes can continue delivering messages. As our proofs deal only the with finiteness of storage space, and not how it is apportioned, we hereafter assume $l = 0$. From Agreement we obtain,

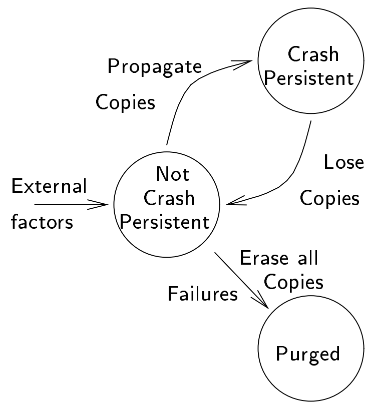

Figure 2: Persistency transitions of a message.

Definition 5 *A message m is* obsolete *in* S *exactly when every process in* S *either has already delivered m or is crashed,*

$$\textsc{obsolete}(\mathsf{S}, \Phi_m) \stackrel{\text{def}}{=} \bigwedge_{q \in \mathsf{S}} \textsc{deliver}_q(m) \ \vee \ \textsc{crash}(q) \ .$$

Definition 6 *A formula ϕ has been* purged *from* S *if, even though true, it is never distributed knowledge among the non-crashed members of* S*,*

$$\textsc{purged}(\mathsf{S}, \phi) \stackrel{\text{def}}{=} \phi \ \wedge \ \Box \neg D_{\mathsf{nc}(\mathsf{S})} \phi \ .$$

Once a formula is purged from S, it is impossible for any member of S to (re)learn or know it; from Formula 3 it follows that once a message is removed from crash-persistent storage it is possible it may be purged. Figure 2 depicts the persistency transitions of a message. Initially, it is not crash-persistent but can be made so by propagating enough copies of it so that the number of copies stored exceeds the number of crashes that may occur. While it is not crash-persistent, process failures can cause it to be purged, as can erasing all existing copies. A crash-persistent message can become not crash-persistent if some sites erase their copy so that fewer copies remain than the number of failures that may occur.

System Property 4 *Local events are not observable to remote processes.*

Local events include crashing, writing to and reading from local storage, receiving a message, delivering a message, events by which local variables are initially set, and the initial awareness of a message to be reliably broadcast by the process at which it originates (from Unpredictability).

Proposition 2 *If any process p knows Φ_m then either p is the originator of m, or p received a message from o_m implying Φ_m,*

$$K_p\Phi_m \Rightarrow \begin{cases} \neg\text{CRASH}(p) & p = o_m \\ \neg\text{CRASH}(p) \ \wedge \ \text{RECV}_p(o_m, \Phi_m) & p \neq o_m \end{cases}$$

PROOF Follows from Chandy-Misra [7], and System Property 4. In their terminology, the proposition formalizing the fact that a process has taken an un-observable event is *local* to that process. $\boxed{\text{QED}}$

Proposition 3 *If message m is obsolete in S, Φ_m can be removed from crash-persistent storage. Conversely, as long as m is not obsolete in S and some non-crashed process has delivered m, it must remain crash-persistent in S.*

PROOF From Formula 1, $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$ implies Agreement for m, and therefore safety; since $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$ is stable, safety for m can never be violated. For the converse, we will construct a run that violates Agreement if a message that is not obsolete is removed from crash-persistent storage.

1. Define $\mathcal{C}_1 \stackrel{\text{def}}{=} \left\{c_1 \mid c_1 \models \neg\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \wedge \ \text{DELIVER}_p(m) \ \wedge \ \neg\text{CRASH}(p)\right\}$.

   By definition, $\neg\text{OBSOLETE}(\mathsf{S}, \Phi_m) \Rightarrow \exists q \in \mathsf{S} : \neg\text{DELIVER}_q(m) \ \wedge \ \neg\text{CRASH}(q)$.

2. *Claim:* There are cuts in $\mathcal{C}_1$ on which q has never known Φ_m.

   Justification: If there are no such cuts, then in all runs containing a cut in $\mathcal{C}_1$, if $q \neq o_m$ then within the finite period from c_0 to $c_1 \in \mathcal{C}_1$, it is guaranteed that some message from o_m reached q (Proposition 2) which asynchrony and some-chance channels preclude.

   Define $\mathcal{C}_2 \subseteq \mathcal{C}_1$, $\mathcal{C}_2 \stackrel{\text{def}}{=} \left\{c_2 \in \mathcal{C}_1 \mid c_2 \models \boxminus\neg K_q\Phi_m\right\}$.

3. To prove the proposition by contradiction, consider all runs completing cuts in $\mathcal{C}_2$ that also satisfy $\neg\text{C-PERSIST}(\mathsf{S}, \Phi_m)$. That is for $c_2 \in \mathcal{C}_2$, we now restrict attention to

   $\mathcal{R}_1 \stackrel{\text{def}}{=} \left\{r_1 \mid \exists c_2 \in \mathcal{C}_2, (r_1, c_2) \models \Diamond\neg D_{\mathsf{nc(S)}}\Phi_m\right\}$.

4. Further restrict attention to the subset of $\mathcal{R}_1$ in which only p and q are correct (if there are no such runs, then failures are not independent).

   $\mathcal{R}_2 \stackrel{\text{def}}{=} \left\{r_2 \in \mathcal{R}_1 \mid \exists c_2 \in \mathcal{C}_2 : (r_2, c_2) \models \Box\neg\text{CRASH}(\{p, q\}) \wedge \Diamond\text{CRASH}(\mathsf{S} - \{p, q\})\right\}$.

   Of course, runs in $\mathcal{R}_2$ are also runs in which at the relevant c_2, p has delivered m (from $c_2 \in \mathcal{C}_1$), q has never known Φ_m (from $c_2 \in \mathcal{C}_2$), and Φ_m is not crash-persistent (from $r_2 \in \mathcal{R}_1$).

5. For each $r_2 \in \mathcal{R}_2$ let c_{r_2} be the point at which distributed knowledge of Φ_m is lost, $(r_2, c_{r_2}) \models \neg D_{\mathsf{nc(S)}} \Phi_m$. If there are more than one such cut in a run, index them appropriately.

6. *Claim:* Some of the pairs (r_2, c_{r_2}) satisfy $\Box\neg D_{\mathsf{nc(S)}} \Phi_m$.

   Justification: If this is not the case, then Transience is violated (Formula 3). Specifically, m would be guaranteed to recur in all runs of $\mathcal{R}_2$ after each time distributed knowledge of Φ_m were lost.

7. Let $\mathcal{R}_3 \subseteq \mathcal{R}_2$ be the runs in which Φ_m is purged,

$$\mathcal{R}_3 \stackrel{\text{def}}{=} \left\{ r_3 \in \mathcal{R}_2 \mid \exists c_{r_3} \in \mathcal{C}_2 : (r_3, c_{r_3}) \models \Box\neg D_{\mathsf{nc(S)}} \Phi_m \;\wedge\; \text{CRASH}(\mathsf{S} - \{p, q\}) \right\} .$$

   As only p and q are correct in $\mathcal{R}_2$, and $\mathcal{R}_3 \subseteq \mathcal{R}_2$, we have chosen the cut c_{r_3} 'late enough' in r_3 so that both distributed knowledge of Φ_m has been forever lost, and all processes that will crash in r_3 have crashed by c_{r_3}.

8. *Claim:* Some of the pairs (r_3, c_{r_3}) satisfy $\neg\boxminus K_q \Phi_m$.

   Justification: The period between c_2 and c_{r_3} is finite; as such there are some c_{r_3} such that q has not received any some-chance messages between $c_2 \in \mathcal{C}_2$ and c_{r_3} (asynchrony may delay them or they may have been dropped). We have constructed

$$\begin{aligned} \mathcal{C}_3 \stackrel{\text{def}}{=} \Big\{ c_3 \in \mathcal{C}_2 \mid \exists r_3 \in \mathcal{R}_3 : (r_3, c_3) \models\; & \boxminus\neg K_q \Phi_m \;\wedge\; \Box\neg\text{CRASH}(\{p, q\}) \;\wedge \\ & \Box\neg D_{\{\text{p,q}\}} \Phi_m \;\wedge\; \text{DELIVER}_p(m) \Big\} . \end{aligned}$$

9. From $(r_3, c_3) \models \Box\neg D_{\{\text{p,q}\}} \Phi_m \wedge \boxminus\neg K_q \Phi_m$, we conclude $(r_3, c_0) \models \Box\neg K_q \Phi_m$, and therefore $(r_3, c_0) \models \Box\neg\text{DELIVER}_q(m)$ (from Integrity, Formula 2). As p and q are both correct in r_3 and p has delivered m, r_3 violates Agreement. $\boxed{\text{QED}}$

Proposition 3 shows that solutions to RRB can only remove obsolete messages from crash-persistent storage. We assume obsolete messages are removed from crash-persistent storage fairly. If some are not, this reduces the storage resources available and implies some messages are more important than others; for generality, we do not make this distinction.

Proposition 4 *If any process has delivered $(\delta + 1)$ messages, at least one of these messages must be obsolete.*[6]

[6]For the general case, when $l > 0$, the limit is $\delta + l + 1$ messages delivered.

PROOF Since at most δ messages can be in crash-persistent storage simultaneously, the 'pigeon-hole principle' means one of the $(\delta + 1)$ messages is not crash-persistent. Proposition 3 says this message must be obsolete. $\boxed{\text{QED}}$

Applying knowledge-generalization to Propositions 3 and 4 means that when any process p is about to deliver its $(\delta + 1)^{th}$ message, it must know that one of the previous messages is obsolete. From the definition, we have,

$$K_p\text{OBSOLETE}(\mathsf{S}, \Phi_m) \;\Leftrightarrow\; \bigwedge_{q \in \mathsf{S}} K_p\Big(\text{DELIVER}_q(m) \;\vee\; \text{CRASH}(q)\Big)\ .$$

5 Failure Detectors and RRB

In this section, we show that knowing a message is obsolete requires a process to have, and to know it has, a *perfect failure detector* (one that makes no mistakes in suspecting process and eventually continually suspects all crashed processes) [5]. We prove that when message storage space is full, and one message must be rendered not crash-persistent, every process must be precisely accounted for regarding its knowledge of the message that is about to be overwritten, or its failure. We then prove only perfect failure detectors (not even eventually perfect failure detectors) provide knowledge of failures. We begin with a brief description of failure detectors (see [5] for a complete treatment).

5.1 Failure Detector Definitions

Chandra and Toueg defined classes of *failure detectors*, which can be thought of as supplemental functionality for an asynchronous distributed system, and proved that a very weak class of failure detector is necessary and sufficient for solving Distributed Consensus [4]. In their model, a failure detector is associated with each process and notifies the process about suspected failures of remote processes. All processes possess failure detectors with the same properties. Denote p's failure detector by $\mathcal{F}_p$. The suspicion of a failure does not alter the behavior of a process toward the suspected process; for example, if p suspects q in round i of a protocol, it need not suspect q in round $i + 1$ unless $\mathcal{F}_p$ again indicates it should.

Failure detectors possess both accuracy and completeness properties; an accuracy property restricts the false detections (or erroneous suspicions) a failure detector can emit.

Strong Accuracy: No correct process is ever suspected by any $\mathcal{F}_p$.

Weak Accuracy: Some correct process is never suspected by any $\mathcal{F}_p$.

Eventual Strong Accuracy: Each $\mathcal{F}_p$ eventually attains Strong Accuracy; before then, its behavior is unrestricted.

Eventual Weak Accuracy: Each $\mathcal{F}_p$ eventually attains Weak Accuracy; before then, its behavior is unrestricted.

The moment at which $\mathcal{F}_p$ attains Strong (or Weak) Accuracy is not known or determinable. A completeness property describes the extent to which crashed processes must be permanently suspected.

Strong Completeness: Every crashed process is eventually continually suspected by all correct processes.

Weak Completeness: Every crashed process is eventually continually suspected by some correct process.

Failure detector classes are defined according to which completeness and which accuracy property they possess.

	Strong Acc'y	Weak Acc'y	Eventual SA	Eventual WA
Strong Comp	$\mathcal{P}$ (perfect)	$\mathcal{S}$ (strong)	$\diamond\mathcal{P}$	$\diamond\mathcal{S}$
Weak Comp	$\mathcal{Q}$	$\mathcal{W}$ (weak)	$\diamond\mathcal{Q}$	$\diamond\mathcal{W}$

5.2 Knowing $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$

Proposition 5 follows from the nature of local events. To keep the discussion simple, we consider only direct communication between processes, and do not explicitly model forwarding information through other processes.

We (ab)use the event $deliver_q(m)$ below to represent the contents of any some-chance message q sends another process informing it that q has delivered m.

Proposition 5 *Knowledge of reliably delivering a message is local to a process,*

$$K_p\text{DELIVER}_q(m) \Rightarrow \text{RECV}_p(q, \text{deliver}_q(m)) \ .$$

PROOF (à la Chandy-Misra [7]) Similar to Proposition 2, the formula $\text{DELIVER}_q(m)$ is local to process q. $\boxed{\text{QED}}$

Proposition 6 strengthens Proposition 3 by considering the knowledge a process must have before overwriting a local copy of a stored message. While Proposition 3 states that any obsolete messages may safely be removed from crash-persistent storage, Proposition 6 in fact restricts individual processes from making obsolete messages not crash-persistent. Specifically, Proposition 6 says it is impossible for a process to know a message is obsolete without knowing definitively whether another process has delivered the message or has crashed.

Proposition 6 *If a process p knows a message m is obsolete, then it can account definitively for every other process q either delivering m or being crashed,*

$$K_p\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \Rightarrow \ \bigwedge_{q \in \mathsf{S}} K_p\text{DELIVER}_q(m) \ \vee \ K_p\text{CRASH}(q) \ .$$

PROOF By way of contradiction, consider any cut in any run on which p knows Φ_m is obsolete, but does not know that q knew Φ_m and does not know that q is crashed,

$$c \models K_p\Big(\text{DELIVER}_q(m) \ \lor \ \text{CRASH}(q)\Big) \ \land \ \neg K_p\text{DELIVER}_q(m) \ \land \ \neg K_p\text{CRASH}(q) \ .$$

Given p's lack of knowledge, the following four cases are combinatorially (though as we show, not all fundamentally) possible:

1. There are $\sim_p$ cuts that satisfy $\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \land \ \text{DELIVER}_q(m) \ \land \ \text{CRASH}(q)$.
2. There are $\sim_p$ cuts that satisfy $\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \land \ \text{DELIVER}_q(m) \ \land \ \neg\text{CRASH}(q)$.
3. There are $\sim_p$ cuts that satisfy $\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \land \ \neg\text{DELIVER}_q(m) \ \land \ \neg\text{CRASH}(q)$.

   Such cuts cannot exist.
4. There are $\sim_p$ cuts that satisfy $\text{OBSOLETE}(\mathsf{S}, \Phi_m) \ \land \ \neg\text{DELIVER}_q(m) \ \land \ \text{CRASH}(q)$.

   Let d be such a cut, $d = (h_1, ...h_q, ...h_p, ...h_n)$ with $h_q = start_q \cdot e_q^1 \cdots e_q^k \cdot crash_q$. Now consider h'_q, which is exactly the same as h_q except that q has not yet crashed, $h'_q = start_q \cdot e_q^1 \cdots e_q^k$. Construct the cut $d' = (h_1, ...h'_q, ...h_p, ...h_n)$. In both h_q and h'_q, q sends no messages after the event e_q^k and $crash_q$ is unobservable to p. That is, $d' \sim_p d \sim_p c$, so $d' \models \text{OBSOLETE}(\mathsf{S}, \Phi_m)$. As d' is a prefix of d, it also satisfies $\neg\text{DELIVER}_q(m)$. That is, $d' \models \neg\text{DELIVER}_q(m) \ \land \ \neg\text{CRASH}(q)$, which means d' cannot satisfy $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$.

   As in the third case, such cuts cannot exist.

Because the last two cases are impossible, whenever $c \models K_p\text{OBSOLETE}(\mathsf{S}, \Phi_m)$, every $\sim_p$ cut also satisfies $\text{DELIVER}_q(m)$. That is,

$$K_p\text{OBSOLETE}(\mathsf{S}, \Phi_m) \Rightarrow \bigwedge_{q \in \mathsf{S}} K_p\text{DELIVER}_q(m) \Rightarrow \bigwedge_{q \in \mathsf{S}} \text{RECV}_p(q, deliver_q(m)) \ .$$

Consequently, there is no solution to RRB if even a single process can crash (it may crash before delivering a message or telling others it has delivered a message). However, a process can know a message is obsolete in such runs provided it can know crashed processes are, indeed, crashed. That is, for all runs and all cuts it is not possible for a process to know a message is obsolete without knowing how each process has satisfied the defining disjunction,

$$\neg\Big(K_p\text{OBSOLETE}(\mathsf{S}, \Phi_m) \land \Big(\exists q \in \mathsf{S} : \neg K_p\text{DELIVER}_q(m) \ \land \ \neg K_p\text{CRASH}(q)\Big)\Big) \ .$$

$\boxed{\text{QED}}$

Proposition 6 means that unless a process can account for every other process definitively, it cannot know $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$ and therefore cannot take any action that will render Φ_m not crash-persistent. Together with Proposition 5, this means a process must query each member of S regarding the delivery of m before it can know $\text{OBSOLETE}(\mathsf{S}, \Phi_m)$. Furthermore, a process cannot overwrite its local copy until it knows any unresponsive process is crashed. The next propositions phrase (one aspect of) the failure detector hierarchy in terms of knowledge of process failures. Certainly knowledge of failures follows from strong accuracy (no non-crashed process is suspected), but we show that no amount algorithmic cleverness can produce knowledge of failures from imperfect failure detectors.

5.3 Failure Detectors and Knowledge of Failures

Proposition 7 *Given only unreliable failure detectors, no process can ever know another process is crashed.*

PROOF We prove that if this were the case, unreliable failure detectors and some-chance channels could implement perfect failure detectors. Given $\mathcal{W}$, the class of weak failure detectors, any member satisfies Weak Accuracy and Weak Completeness. Now let $\mathcal{W}^+$ be the class of failure detectors that possess, in addition to the properties of $\mathcal{W}$, the guarantee that each crashed process is eventually *known* to be crashed by some correct process. The following algorithm constructs a perfect failure detector from any member of the class $\mathcal{W}^+$, using only some-chance channels.

```
/* process p, with failure detector F_p ∈ W+ */
Know_p = ∅;
while (true) do {
    ∀q ∈ S − Know_p {
        if recv_p(q, Know_q) and r ∈ Know_q
            then Know_p = Know_p ∪ {r}
        else if F_p says knows(q)
                then Know_p = Know_p ∪ {q}
            else if F_p says suspect(q) then skip;
    }
    ∀q ∈ S − Know_p
        send_p(q, Know_p);  /* line 11 */
}
```

Take Know_p as the output (*i.e.*, suspicion list) of a perfect failure detector. Only crashed processes are ever in Know_p, so Strong Accuracy holds. For Strong Completeness, let $\mathsf{C} \subseteq \mathsf{S}$ be the set of correct processes (in a given run). Between them, their $\mathcal{W}^+$ failure detectors ensure each of the crashed processes is eventually known to be crashed. Each time Know_p changes, p sends an infinite number of messages whose contents imply the new Know_p (line 11). This is because processes are never removed from Know_p. From Proposition 1, eventually every other $q \in \mathsf{C}$ receives a message from p mentioning $r \in \mathsf{Know}_p$, and adds r to its own Know_q. Similarly, eventually p receives every other Know_q and merges these with Know_p. That is, strong completeness holds because for each $p \in \mathsf{C}$, eventually every crashed process is in Know_p. $\boxed{\text{QED}}$

Proposition 8 *Given only eventually reliable failure detectors, no process can ever know another process is crashed.*

PROOF Eventually reliable failure detectors cannot give knowledge of process crashes because the instant at which they become reliable, while guaranteed to occur, cannot be ascertained. That is, even if it knows its failure detector is in the class $\diamond\mathcal{P}$, a process must always consider that its failure detector is not yet reliable. The complete proof is similar to the proof of Proposition 7: an enhanced $\diamond\mathcal{P}^+$ failure detector that gives knowledge of failures can be shown to implement a member of $\mathcal{P}$. $\boxed{\text{QED}}$

Proposition 9 *In the absence of infinite storage capacity, unreliable failure detectors cannot be used to solve RRB if even one process can fail.*

PROOF Any solution to RRB must overwrite stored messages if more than δ messages are to be delivered, and only obsolete messages can be intentionally overwritten (Proposition 3). To ensure safety, a process p must know Φ_m is obsolete before overwriting (and possibly purging) it. In runs in which some process q fails before delivering the message, Proposition 6 requires p to know q is crashed, which Proposition 7 proves is impossible for unreliable failure detectors. $\boxed{\text{QED}}$

Proposition 10 *In the absence of infinite storage capacity, eventually reliable failure detectors cannot be used to solve RRB if even one process can fail.*

PROOF The proof is similar to that of Proposition 9 and follows from Proposition 8. $\boxed{\text{QED}}$

5.4 Awareness of Perfect Failure Detectors

Even if its failure detector is perfect, p can only infer $\text{CRASH}(q)$ (making $K_p\text{CRASH}(q)$ true) if p knows it has a perfect failure detector.[7] It is commonly assumed that processes know which class their failure detector belongs to. We show perfect failure detectors suffice to solve RRB. For Proposition 9 and Proposition 10, knowledge of the type of failure detector does not help, as the failure detectors considered are still too weak to give knowledge of failures.

Proposition 11 *In the absence of infinite storage capacity, perfect failure detectors are necessary and sufficient for solving RRB if* $0 < f < |\mathsf{S}|$ *processes can fail.*

[7]As in the proof of Proposition 8, not knowing when a $\diamond\mathcal{P}$ failure detector becomes perfect precludes $\diamond\mathcal{P}$ from giving knowledge of process failures.

Proof For sufficiency, the algorithm below uses some-chance channels and a perfect failure detector to purge obsolete messages.

/* p overwriting Φ_m in msg-store$_p$, with failure detector $\mathcal{F}_p \in \mathcal{P}$. */
$\forall q \in \mathsf{S}$ {
  repeat
    $send_p(q, RequestPurge(\Phi_m))$
  until $\mathcal{F}_p$ emits $suspect(q)$ or $recv_p(q, deliver_q(m))$
}
/* $\forall q \in \mathsf{S} : K_p\text{CRASH}(q) \vee K_p\text{DELIVER}_q(m)$. */
erase local copy of Φ_m.

For necessity, Strong Accuracy guarantees that suspicions are never wrong; if p suspects q, then $K_p\text{CRASH}(q)$, which Proposition 6 shows is necessary before rendering Φ_m not crash-persistent. Until p suspects q, it must continue to wait for a message from q indicating $\text{DELIVER}_q(m)$. Without Strong Completeness, if q is crashed, p may wait forever. QED

5.5 Generalizing the Results

It should be clear that the need for a perfect failure detector does not depend on the fact that we have chosen to define the problem in terms of repeating Reliable Broadcast. The need for perfect failure detectors is attributable to the need to do something infinitely often, the need to keep this information for all processes that never crash, and the reality of finite storage space. We define *Generic Repeated Coordination* (GRC), and show the results for RRB hold for GRC.

Definition 7 (Generic Repeated Coordination) *Given the obligation set,* S*, let* A *be an infinite name space of actions that processes execute with the special, local event* $\text{do}_p(a)$ *for* $p \in \mathsf{S}, a \in A$.

Unpredictability *Actions to be taken become known to members of* S *but (Input) external factors dictate which actions will arise in any run and (Location) the member of* S *at which a particular action arises is not known in advance.*[8]

Transience *Any non-local factors responsible for generating an action may not recur.*

Integrity *The obligation to execute* $\text{do}_p(a)$ *only applies if* a *arises at a member of* S.

Agreement *If any correct process* p *in* S *executes the event* $\text{do}_p(a)$ *then every other correct process* q *in* S *eventually executes* $\text{do}_q(a)$.

Validity *A correct process takes all actions that arise at it.*

[8] An action can arise at a subset of S, too.

Unboundedness *In any run, the number of actions that will arise at a correct process is not known or fixed in advance.*

Instead of storing messages they have delivered, processes store actions they have taken.

Proposition 12 *Any instance of Generic Repeated Coordination requires either a perfect failure detector or infinite-capacity crash-persistent storage if even one process in* S *can crash.*

PROOF Suppose $|\mathsf{S}| > f > 0$ processes can crash and that at most δ different actions can be kept in crash-persistent storage simultaneously. Transience means information about actions must be migrated to crash-persistent storage to ensure all correct processes can learn it to satisfy Agreement. Unboundedness and Validity require an unpredictable number of actions to be stored. Generalizing Proposition 4, at least $(i - \delta)$ must be obsolete and purged when any process takes its i^{th} action. Only Agreement (safety) can define OBSOLETE(), and it exempts only crashed processes from having to take actions. Proposition 6 shows that no process can overwrite a local copy of information about an action unless it has accurate knowledge of process crashes.

If, on the other hand, the system has infinite storage capacity, all actions that arise can be kept crash-persistent. Since there is no need to purge information, there is no need for accurate knowledge of process crashes. $\boxed{\text{QED}}$

Thus, the particular coordination problem being solved repeatedly only affects the number of copies needed to make decision determinants crash-persistent. In this way, the power of an adversary to crash processes is determined by the underlying distributed coordination problem, not the fact that the problem is being solved repeatedly. Proposition 12 shows that irrespective of the inherent fault tolerance of the problem, as long as the Agreement condition only exempts crashed processes from participating, and storage is finite, perfect failure detectors are necessary.

6 Avoiding Reliable Channels and Reliable Broadcast

We now describe conditions under which some-chance channels suffice. Consider that a channel between p and q can buffer B different messages simultaneously (*e.g.*, any sliding window protocol). After attempting to send B different messages, p must either overwrite one of the messages buffered, or not send another message until q acknowledges receiving one of the pending messages. However, if the contents of the message p overwrites is contained in, or implied by, an infinite number of messages that p will send in the future, there should be no problem. Obviously, messages' sizes should not grow infinitely; for simplicity, we assume each message occupies at most one buffer slot.

Definition 8 *A sequence of messages from process p to process q is* monotonic *if and only for each message m_i there are an infinite number of messages in the sequence whose contents logically imply the contents of m_i, and the size of each messages is bounded by some constant b.*

A consequence of the boundedness condition is that only a finite number of messages in a monotonic sequence can have distinct contents. Proposition 13 shows that if p and q are correct and the message sequence from p to q is monotonic, some-chance channels guarantee that q will eventually learn everything p has been trying to send it. However, if the message sequence is not monotonic, reliable channels, and therefore perfect failure detectors, are needed.

Proposition 13 *If p and q are correct, and $m_1, m_2, \ldots$ is a monotonic messages sequence from p to q over a some-chance channel, then for any m_i, eventually q receives a message with the same contents as m_i. Conversely, if the sequence is not monotonic, the channel must be reliable.*

PROOF Let $M_i \stackrel{\text{def}}{=} \{m_j \mid i < j, \Phi_{m_j} \Rightarrow \Phi_{m_i}\}$. As the sequence is monotonic, all m_i are of bounded size and each M_i contains infinitely many messages. From Proposition 1, q receives at least one message in each M_i.

If messages are not monotonic, then there is some M_i that has only finite cardinality, $n_i < \infty$. If ϵ is the probability than a given message is successfully transmitted, the probability that no message in M_i is successfully transmitted is $(1-\epsilon)^{n_i}$, which does not approach zero for finite n_i; some-chance channels cannot guarantee that q will eventually receive a message with the same contents as m_i. $\boxed{\text{QED}}$

6.1 Solving Consensus with $\mathcal{S}$

Consequently, the algorithm given in [5] that uses a member of $\mathcal{S}$ to solve Consensus actually depends on reliable channels, not some-chance channels, because the sequence of messages any process p sends to the other processes is not monotonic; in each round, p sends only its change vector, Δ_p, which does not imply any information about previous rounds and messages. A full-information protocol would not have messages of bounded size. If p instead sends its cumulative vector, V_p, in each round rather than Δ_p, the message sequence is monotonic.

6.2 Implementing $\mathcal{S}$ from $\mathcal{W}$

While knowledge of failures separates Weak from Strong Accuracy (not surprisingly), it does not separate Weak from Strong Completeness. Said another way, reliable channels are not needed to implement a member of $\mathcal{S}$ from a member of $\mathcal{W}$ (or a member of $\diamond\mathcal{S}$ from a member of $\diamond\mathcal{W}$). This should be clear from the proof of Proposition 7.

In the terminology of Proposition 12, the problem of Strong Completeness can be cast to incorporate Unpredictability (here 'actions' are permanent suspicions), Integrity, Agreement, and Validity. However, given Weak Completeness, it does not incorporate Transience, because the 'actions' are guaranteed to arise continuously from an external source (in this case the $\mathcal{W}$, or $\mathcal{W}^+$, or $\diamond\mathcal{W}$failure detector). Neither does it incorporate Unboundedness since at most $n-1$ processes can crash. The problem of Strong Completeness (given Weak Completeness) is not an instance of GRC.

Proposition 14 *Given Weak Completeness, some-chance channels suffice to implement Strong Completeness.*

PROOF Suppose q crashes and p is a correct process that permanently suspects q under Weak Completeness. Since q can only have sent finitely many messages, eventually every correct r receives a final message from q. In the Chandra and Toueg algorithm, and ours in Proposition 7, once p permanently suspects q, the messages it sends informing others of these suspicions are monotonic regarding q's faultiness. Proposition 1 means every correct r will receive an infinite number of suspicions from p, and therefore an infinite number *after* having received its last message from q. In both algorithms, if p's suspicions are received, they are adopted. Finally, once r has received its last message from q, and subsequently suspected q, it cannot stop suspecting q. $\boxed{\text{QED}}$

6.3 Consensus and Atomic Broadcast

We have shown how to fix Chandra and Toueg's solution to Consensus using a member of $\mathcal{S}$ so that some-chance channels suffice, and we have shown that some-chance channels can implement a member of $\mathcal{S}$ (respectively $\diamond\mathcal{S}$) from a member of $\mathcal{W}$ (respectively $\diamond\mathcal{W}$). We have shown that Consensus can be solved with some-chance channels and a $\mathcal{W}$ (or $\diamond\mathcal{W}$) failure detector. We have:

Proposition 15 *In the absence of infinite storage capacity, Consensus is weaker than RRB if* $0 < f < |\mathsf{S}|$ *processes can fail.*

However, the same is *not* true for Atomic Broadcast, which is RRB with an additional *Total Order* requirement: if two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m' [5]. Usually, Consensus is used to handle the Total Order requirement, with RRB taken for granted. In contrast to Consensus, some-chance channels will not suffice for Atomic Broadcast as the information on messages and their delivery order cannot be monotonic. For example, in Chandra and Toueg's algorithm, the consensus decision that determines the set of messages to be *A-delivered* in the k^{th} batch must be crash-persistent;[9] once it is overwritten, a correct process cannot learn which messages constituted the k^{th} batch. Thus, we have the equivalence of RRB and Atomic Broadcast.

Proposition 16 *In the absence of infinite storage capacity, Atomic Broadcast and RRB are equivalent if* $0 < f < |\mathsf{S}|$ *processes can fail.*

PROOF Chandra and Toueg's algorithm in [5] uses a solution to RRB and, in our work the weaker problem, Consensus to solve Atomic Broadcast; by definition any solution to Atomic Broadcast is a solution to RRB. $\boxed{\text{QED}}$

[9]Delivery order for the messages within each batch is determined by an *a priori* commonly-known function.

7 Discussion and Conclusions

We have shown that any coordination problem that only exempts crashed processes from taking the coordinated actions, and that must be solved repeatedly requires either a perfect failure detector or the ability to store infinitely many of the coordinated actions. Proposition 13 does not contradict Proposition 13 because Unboundedness in GRC generally ensures that no message sequence will be monotonic. As Proposition 12 depends solely on accurate failure detection and finite storage, our results are not sensitive to how processes communicate with each other; in particular, the result can be extended to non-message-passing systems.

Lacking perfect failure detectors, future work can focus on appropriate weaker exemptions from coordination. Specifically, the need to know $\text{CRASH}(q)$ before purging information from crash-persistent storage should be replaced by something a process in S can actually learn. For many applications' consistency, it should also be something that is indistinguishable, certainly to clients of any service S is implementing, from q having crashed. There has been some work considering GRC with broader exemptions that are indistinguishable from $crash_q$ [17, 18]. Sabel and Marzullo have taken a similar approach in simulating failure detectors [19].

It is often said that the inability to know whether a process is crashed is at the heart of the FLP impossibility result for Distributed Consensus. In fact, our results show this explanation is incorrect. Surely if one could know whether another process were crashed (*i.e.*, if one had a perfect failure detector) one could solve Consensus, but unreliable failure detectors suffice, even with some-chance channels. One possible explanation is that Weak Accuracy already provides a very primitive consensus construct: there is *a priori* consensus among all (and therefore all correct) processes never to suspect some correct process. This anonymous consensus structure is used to bootstrap the Consensus decision by discovering the identity of the highest-ranked process that has never (yet) been suspected. This leaves an interesting question: what exactly are asynchronous systems lacking that makes Consensus impossible? Our results show it is not knowledge of process failures, and the hidden Consensus construct provided by Weak Accuracy seems to answer the question recursively.

Closer inspection of Weak Accuracy reveals a more fundamental conundrum. Weak Accuracy requires that some correct process, $\hat{p}$, is never suspected by any process. For algorithms that rely on Weak Accuracy, liveness is only achieved if every message sent by $\hat{p}$ is eventually received. Said another way, in assuming Weak Accuracy, it appears that applications are assuming reliable channels from $\hat{p}$ to every other process (and therefore Strong Accuracy for each $\mathcal{F}_p$ regarding $\hat{p}$). This suggests the possibility of collapsing the failure detector hierarchy when storage space is considered in the computing model.

We close by remarking that GRC is a combination of the following conditions.

Persistence The solution ensures that all relevant information will be available to any process.

Blocking The solution does not allow any process to act unless each obliged process either has the needed information or is known to be exempt.

Our results indicated that all solutions embody a combination of the two. Specifically, when there is no storage space, Persistence is Blocking; conversely, Persistence has no element of Blocking only if storage space is infinite.

Acknowledgments

Rachid Guerraoui and André Schiper contributed many valuable comments. Gustavo de Veciana helped formalize Proposition 1.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., 1987.

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

[3] B. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Jorunal of the ACM*, 32(4):824–840, 1995.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *11th PODC*, pages 147–158. ACM, 1992. (also Cornell University TR94-1426).

[5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Tenth PODC*, pages 325–340. ACM, 1991. (also Cornell University TR93-1374).

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

[7] K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*, 1(1):40–52, 1986.

[8] J. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, 1984.

[9] D. E. Comer. *Internetworking with TCP/IP (Volume I).* Prentice Hall, 1991. 2nd Edition.

[10] A. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In *3rd WDAG. Springer Verlag (LNCS 392)*, pages 110–123, 1989.

[11] V. Hadzilacos and S. Toueg. *Distributed Systems (2nd Edition). Edited by S. Mullender*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Adison-Welsey, 1993.

[12] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *JACM*, 37(3):549–587, 1990.

[13] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Message Systems Using Optimistic Message Logging and Checkpointing. In *Seventh ACM Principles of Distributed Computing*, pages 171–181, 1988.

[14] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

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

[16] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmaann, 1994.

[17] D. Malki, K. P. Birman, A. Ricciardi, and A. Schiper. Uniform Actions in Asynchronous Distributed Systems. In *13th PODC*, pages 274–284. ACM, 1994. (also Cornell University TR94-1447).

[18] A. Ricciardi. Dissecting Distributed Coordination. In *9th WDAG. Springer Verlag (LNCS 972)*, pages 101–115, 1995.

[19] L. Sabel and K. Marzullo. Simulating Fail-Stop in Asynchronous Distributed Systems. In *13th Symposium on Reliable Distributed Systems*, pages 138–47. IEEE, 1994.

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

[21] M. D. Skeen. *Crash Recovery in a Distributed Database System.* PhD thesis, University of California at Berkeley, May 1982.