Impossibility of (Repeated) Reliable Broadcast

Aleta Ricciardi

TR TR-PDS-1995-016 September 1995

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

Abstract

We consider the problem of *Generic Repeated Coordination*, of which Reliable Broadcast and Repeated Distributed Consensus are but two instances. We show that in the absence of infinite storage capacity, all problems in this class require *perfect failure detectors*, highlighting significant assumptions made by many protocols assuming reliable broadcast. Our work also shows that the usual reason used to explain the impossibility of Distributed Consensus in asynchronous systems is misattributed; that is, it is not the inability of a process to know whether a remote process is crashed that makes Consensus impossible. Finally, 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", and for a thorough investigation of coordination problems in which the set of participating processes is dynamic.

1 Introduction

Agreement and coordination problems in distributed systems have been researched extensively (Hadzilacos and Toueg have compiled a large survey [12]), 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 generalize a range of distributed coordination problems as processes in some known set, S, 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, 7, 12, 16] all discuss atomic multicast, [1, 17, 22] all treat atomic commit, and [5, 14, 15] cover checkpointing. Our focus is on ensuring infinitely many decisions *can* be made, not on *how* any single decision is reached. While we phrase the problem initially as one of Repeated Distributed Consensus, this is only to simplify the presentation and exploit the established connection between Distributed Consensus and failure detectors. Later we define and show these results hold for *Generic Repeated Coordination*, of which Reliable Broadast, Repeated Uniform Agreement, Atomic Multicast, and Leader Election are concrete examples.

We show that, in the absence of infinite storage capacity, *perfect failure detectors* are needed to solve Repeated Distributed Consensus (RDC), even though eventually-weak failure detectors suffice for solving a single instance of Distributed Consensus [4]. This points out a subtle assumption (that of infinite storage capacity) made in using Consensus to solve Atomic Broadcast [4]. We generalize these results to any problem that requires all correct processes to reach the same set of decisions (possibly in the same order). 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. Finally, our work lays a theoretical foundation for research in process membership services for distributed systems.

Section 2 presents the asynchronous computing model and some important system properties. Section 3 defines Repeated Distributed Consensus and Section 4 discusses the special conditions imposed by having to maintain decision information for every correct process. In Section 5 we prove perfect failure detectors are needed for both Repeated Distributed Consensus and the more generic class of repeated coordination problems. We conclude in Section 6.

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. The system is asynchronous in that neither processes' relative speeds nor message transmission delays have fixed bounds. Processes fail by crashing but otherwise follow their assigned protocol. It is well-known that in such an environment, it is impossible to determine accurately whether a remote process is crashed, whether it is executing very slowly, or whether messages from it are taking excessively long to arrive.

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.

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 message m to q) and $recv_q(p, m)$ (q receives m from p). After each event, a process makes all logical inferences from its new local state. A history for process p, h_p, is a sequence of events beginning with the event $start_p$. A *cut* over S is a tuple of finite process histories, one for each $p \in \mathsf{S}$. We assume inter-event causality [16] and consistent cuts [5] are understood. A *system run* is a tuple of infinite process histories, one for each process in S.[1]

Just as inter-event causality reflects progress made between events, the causal relation between two cuts reflects progress made between global states. Formally, given c and c' over S, $c \leq c'$ if and

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

only if for any process, its history component in c is a prefix (not necessarily a strict prefix) of its history component of in c'. Two cuts c and c' are *indistinguishable* to process p (written $c \sim_p c'$) when p's history component (state) in both is identical.

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, even if it is non-volatile, is unavailable to other processes. As well, no process can store information at, or retrieve information from, a remote process, except through that process.*

2.1 Channel Definitions

Figure 1 illustrates the following defintions. In the figure, a process crash is indicated with an X, solid rays between processes are messages that are received, dashed lines between processes represent messages sent that will not be received, dotted rays are acknowledgements, and shaded rectangular regions indicate the periods a process is unable to send messages.

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

To implement reliable channels, a process will normally buffer outgoing messages and continue to resend them until it receives an acknowledgement from the destination. This is usually taken to mean that if p and q both never crash, every message from p to q will arrive at q (and vice versa). Reliable channels are weaker than omission failures which need not guarantee delivery even if both processes never fail [11]. Sliding-window protocols (*e.g.*, TCP) are the most common way to implement reliable channels. Buffering messages to ensure delivery introduces the notion of *capacity* for reliable channels.

Definition 2 *The* capacity *of a reliable channel is the number of messages that can simultaneously be in transit (sent but not known to be delivered). A reliable channel with capacity c is* full *whenever c messages are in transit.*

Without loss of generality, we assume a full, reliable channel prevents a process from sending new messages. Blocking the sender until a full channel has space maintains the channel's semantics; an implementation that 'overwrites' one of the channel's buffer slots cannot be guaranteed to deliver the overwritten message even though both parties never fail.

Channels *per se* do not embody failure detectors; when a channel is full, a sender either waits, possibly indefinitely, for one of the buffered messages to be acknowledged, or needs recourse to some other mechanism that safely frees it from having to send messages through that channel. A receive event is always blocking; when the channel is (or appears to be) empty, the receiver also needs recourse to a mechanism that safely frees it from having to wait for messages that may never arrive.

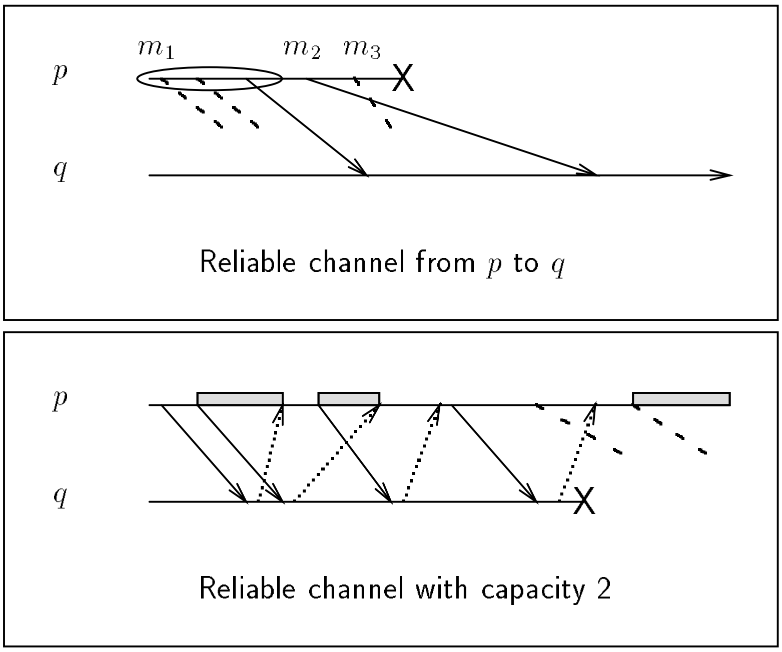

Figure 1: Illustration of channel definitions.

2.2 The Formal Language

We use a logic supporting temporal and knowledge modalities to define the problem. Formulas are evaluated on consistent cuts to better model asynchrony. When formula φ holds on cut c, we write $c \models \varphi$. The modalities have the following semantics:

Time $\Box\varphi$ (always) holds on c if and only if φ holds on c and on all c' such that $c \leq c'$. $\Diamond\varphi$ (inevitably) holds on c if and only if φ holds on *some* $c \leq c'$, in *every* run that includes c. That is, the future is branching time semantics so $\Box$ and $\Diamond$ are not duals.

$\Diamond\!\!\!-\varphi$ (in the past) holds on c if φ holds on some $c' \leq c$. $\boxminus\varphi$ (always in the past) holds on c if φ holds on all $c' \leq c$. Because the past of any cut is contained in it, $\Diamond\!\!\!-$ and $\boxminus$ are duals.

Knowledge [13] $K_p\varphi$ (p knows) holds on c if and only if φ holds on all cuts $c' \sim_p c$. $D_{\mathsf{S}}\varphi$ (distributed knowledge) holds on c if the combined knowledge of all processes in S implies φ.

Crashed processes cannot 'know' anything; that is, $K_p\varphi \Rightarrow \neg\textsc{crashed}(p)$. We remark that System Property 1 is equivalently stated as, if a process is *inevitably* going to crash, it already is crashed, $\Diamond\textsc{crashed}(p) \Rightarrow \textsc{crashed}(p)$.

3 Problem Definition – Repeated Distributed Consensus

Repeated Distributed Consensus (RDC) begins with the simple definition for Uniform Distributed Consensus, which embodies both uniform agreement and termination.[2] Uniform agreement behavior [9, 19] stipulates that if any process makes a decision, every process that never crashes must eventually make the same decision. Fischer *et.al.* defined Distributed Consensus [8], and Guerraoui has shown Uniform Distributed Consensus no harder than Distributed Consensus [10]. Uniformity is desirable if the decision behavior of the processes in S must appear consistent to any process outside S (the original works cited give other justifications). The usual non-triviality and validity conditions apply, though again we are not concerned with how processes reach a decision. To the basic definition, RDC adds the requirement that an unbounded number of consensus decisions be made. The formula $\textsc{decide}_p(x,i)$ means p's i^{th} consensus decision is x, while $\textsc{crashed}(q,i)$ means q crashed before reaching its i^{th} decision. Let S be a fixed set of processes obliged to make decisions together. The safety requirement for the sequential version of RDC states that every process that makes an i^{th} decision, makes the same i^{th} decision,

$$\textsc{decide}_p(x,i) \;\Rightarrow\; \bigwedge_{q\in\mathsf{S}} \Diamond\Big(\textsc{decide}_q(x,i) \;\vee\; \textsc{crashed}(q,i)\Big)\;. \tag{1}$$

[2] While phrased in terms of Consensus, Proposition 9 shows our results apply to all repeated coordination problems with a particular form, including reliable broadcast.

To specify unsequenced RDC we omit the index (if p ever decides x, inevitably every other mutually obliged process either does so as well or crashes),

$$\text{DECIDE}_p(x) \Rightarrow \bigwedge_{q \in \mathsf{S}} \Diamond\Big(\text{DECIDE}_q(x) \ \vee \text{CRASHED}(q)\Big) \ .$$

As the results we present are independent of whether sequence is required, we adopt the sequential notation and problem as it makes refering to specific decisions easier. When necessary we will make comments for the unsequenced case.

Liveness requires that infinitely many decisions can, if necessary, be made,

$$\forall i \geq 1 : \bigvee_{p \in \mathsf{S}} \Diamond \text{DECIDE}_p(*, i) \tag{2}$$

Since we do not (yet) consider an infinite number of processes, or the guaranteed recovery of processes, this means some processes makes infinitely many decisions, though the formula above is more weakly stated.

A *decision determinant*, Φ_i, represents all the information necessary (*i.e.*, without further communication) for a process to infer at any point in an execution the correct i^{th} decision (for the non-sequential case, simply the decision value itself).

System Property 4 *Once a process knows a decision determinant, it can only be prevented from physically making that decision (*e.g.*, committing a transaction, delivering a message, adopting a new leader) if it crashes.*

Knowledge of Φ_i forces a process to decide the value Φ_i encodes, provided the process does not crash, $K_p\Phi_i \Rightarrow \Diamond\Big(\text{DECIDE}_p(x,i) \ \vee \ \text{CRASHED}(p,i)\Big)$. As well, if a process has safely decided on a value, it (inherently) knows the decision determinant, $\text{DECIDE}_p(x,i) \Rightarrow K_p\Phi_i$.

Applying Knowledge-generalization, and a little logical manipulation to Formula 1 gives

$$\text{DECIDE}_p(x,i) \Rightarrow \bigwedge_{q \in \mathsf{S}} K_p\Diamond\Big(K_q\Phi_i \ \vee \ \text{CRASHED}(q,i)\Big) \ . \tag{3}$$

In every solution there must be agreement, by the time any process enters a decision state, on a decision determinant for that decision. Usually, this is called "locking" the value. A solution that is tolerant to f failures makes sure locked values persist and are available to all non-crashed processes despite the failure of f members of S. We next analyze whether such persistence is attainable.

4 Crash Persistence

Definition 3 *A formula ϕ is* crash-persistent *in* S *if it is stable, distributed knowledge among the (non-crashed) process of* S*:*

$$\textsc{c-persist}(\phi) \stackrel{\text{def}}{=} \left(D_{\mathsf{S}}\phi \Rightarrow \Box D_{\mathsf{S}}\phi\right) .$$

Once a process p knows a decision determinant, Φ_i is crash-persistent, then p does not have to know which of the disjuncts in Formula 3 will hold; p knows the information will be available to any process that needs it, despite failures of members of S.[3] If p does not know whether Φ_i is crash-persistent, and it is possible that knowledge of Φ_i may be lost forever, then the Accountability Theorem of [20] holds and p must be able to account, definitively, for every other process q. In fact, this issue arises later (Section 4.2) when some decision determinants are removed from crash-persistent storage to make room for others; we explain this in more detail in the conclusions. The rest of this section focuses on making a decision determinant persist despite process crashes and the passage of time.

4.1 Implementing Crash-Persistent Information

When only crashed processes are exempt from deciding, every process that never crashes must have access to the decision information, no matter how late in the computation and no matter how many other processes have crashed. Lacking an external, fault-tolerant entity, making information crash-persistent means replicating it at the appropriate number of processes to be stored locally, though our Proposition 9 shows our results hold in the presence of common continuously-available storage space that exists independently of any process (*i.e.*, shared memory systems).

Often, channels are used to store information and we have included these in our considerations. Without loss of generality, we assume a process only uses its channel buffering capacity to transmit decision determinants reliably, not for extra storage.

Given f as an *a priori* limit on the number of crashes, safety (Formula 3) means no solution can allow a process to enter a decision state until it *knows* at least $f+1$ processes of S are storing that decision determinant. With reliable channels, p can make Φ_i crash-persistent despite f failures by sending it to f other processes. However, p does not know $\textsc{c-persist}(\Phi_i)$ until it has received acknowledgements from f other processes.

In general, let b the the number of different decision determinants a process can store locally, and let f be the desired degree of crash persistence for any decision determinant. For example, Distributed Consensus using a $\diamond\mathcal{W}$ failure detector has solutions provided a majority of the original set, S, never crash. If S has n members, the total available storage over S is $A = nb$ and the number of decision determinants that can be crash-persistent at any point is $\delta = \lfloor A/f \rfloor$.

[3] It is also vital for safety that p know q will be able to use Φ_i to make its i^{th} decision.

4.2 Purging Crash-Persistant Information

For higher level discussions, we abstract crash-persistent storage as δ separate bins that can each store one decision determinant. In this abstraction, any non-crashed process can always read the contents of any bin (this is what crash-persistent means), and the contents of any box is changed atomically (a formula either is, or is not, crash-persistent); when a formula is not crash-persistent, a process may learn it but cannot rely on others doing so. Suppose all bins are full, containing Φ_1, Φ_2, ..., Φ_δ, and consider any Φ_i stored. If q is a process that should be aware of this decision, but is not, there is a dilemma (see Figure 2):

- Keeping Φ_i crash-persistent prevents any process from making a $(\delta+1)^{th}$ decision. Without space to make the new decision determinant crash-persistent, we have a liveness violation.
- On the other hand, if Φ_i is overwritten to store $\Phi_{(\delta+1)}$, all knowledge of the actual decision (whether sequenced or not) is lost. For the sequenced case, it can thereafter never be verified whether q will make the correct i^{th} decision. For the unsequenced case, it cannot be guaranteed q will inevitably learn about the decision value. In either case, unless q can be guaranteed to crash, safety is violated.

Consequently we must understand when it is safe to remove a decision determinant from crash-persistent storage (*i.e.*, overwrite it) so that more than δ decisions can be made. The agreement condition (Formula 3) gives a hint.

Definition 4 *A decision determinant, Φ_i, is* obsolete *exactly when every process obliged to make that decision either already knows Φ_i, or will inevitably crash (without making an i^{th} decision),*

$$\text{Not Sequenced: } \textsc{obsolete}(\Phi_i) \stackrel{\text{def}}{=} \bigwedge_{q \in \mathsf{S}} K_q \Phi_i \;\vee\; \Diamond \textsc{crash}(q) \; . \tag{4}$$

$$\text{Sequenced: } \textsc{obsolete}(\Phi_i) \stackrel{\text{def}}{=} \bigwedge_{q \in \mathsf{S}} K_q \Phi_i \;\vee\; \Diamond \textsc{crash}(q, i) \; . \tag{5}$$

Definition 5 *A formula ϕ has been* purged *from S if, even though true, it is impossible for any member of S to know ϕ in the future,*

$$\textsc{purged}(\mathsf{S}, \phi) \stackrel{\text{def}}{=} \phi \;\wedge\; \Big(\bigwedge_{p \in \mathsf{S}} \Box \neg K_p \phi \Big) \; .$$

Figure 3 depicts the life-cycle of a decision determinant. Initially, it is not crash-persistent, but can be made so by propagating enough copies of it so that the number of copies exceeds the number of crashes that may still occur. While it is not crash-persistent, process failures can cause it to be purged, as can erasing all remaining copies. A crash-persistent decision determinant can become not crash-persistent if some sites erase their copy so that fewer copies remain than the number of failures that may still occur.

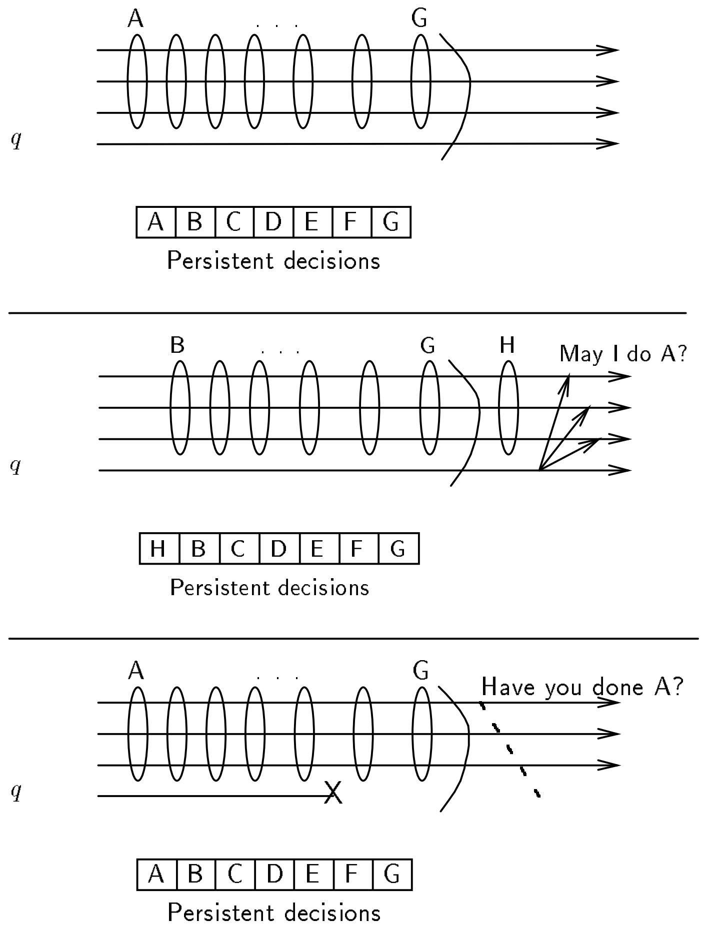

Figure 2: When storage for persistent information is full (top), either safety (middle) or liveness (bottom) may be violated.

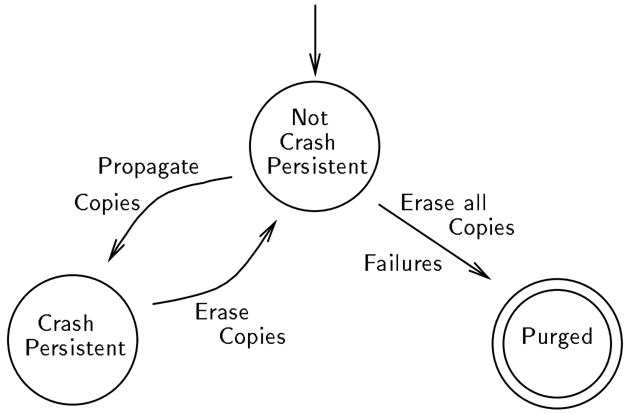

Figure 3: Life-cycle of a decision determinant.

System Property 5 *Neither crashing nor overwriting a local copy of a decision determinant are observable to remote processes.*

System Property 6 *Purging a decision determinant (overwriting local copies, possibly coupled with inopportune random crashes) does not cause any process to crash.*

System Property 7 *Purging a decision determinant (overwriting local copies, possibly coupled with inopportune random crashes) does not cause any process to learn it.*

Proposition 1 *If a decision determinant is obsolete, it can be purged from* S. *Conversely, one that is not obsolete must remain crash-persistent within* S.[4]

PROOF OBSOLETE(Φ_i) implies safety (Formula 1), and since it is stable, safety for that decision can never be violated. Conversely, if the decision determinant is not crash-persistent, the decision value can be irretrievably lost due to crashes. If it is not obsolete, some process exists that neither knows it, nor has crashed. That process is required to reach the same decision but cannot be guaranteed to do so, nor can it be guaranteed to crash. $\boxed{\text{QED}}$

Any solution must therefore ensure that only obsolete determinants are purged, and that no process actually makes a decision until after that decision determinant is crash-persistent. We assume every obsolete determinant is inevitably purged. If some are not, this reduces the resources available to make new determinants crash-persistent and implies some decision determinants are more important than others; for generality, we do not make this distinction.

[4] A decision determinant that is obsolete may be purged or simply made not crash-persistent; this depends on whether there is room (whether A/f has any remainder) or even the desire to maintain non-crash-persistent copies of the information. We assume there is neither.

Proposition 2 *If any process has made its $(\delta+1)^{th}$ decision, at least one of the first $(\delta{+}1)$ decisions must be obsolete.*

Proof If none of the previous decisions are obsolete then, since only δ decision determinants can be crash-persistent, the pigeon-hole principle means one of the $(\delta+1)$ decision determinants cannot be. This contradicts Proposition 1. $\boxed{\text{QED}}$

It follows simply that if any process reaches its $(\delta+i)^{th}$ decision, at least i of the first $(\delta+i)$ decisions must be obsolete. Applying knowledge-generalization to Proposition 1 and Proposition 2 means that when a process is about to make its $(\delta+1)^{th}$ decision, it must not only know the new determinant is crash-persistent, it must also know one of the previous determinants is obsolete. From the defintion of OBSOLETE(), we have

$$K_p\textsc{obsolete}(\Phi_i) \;\Leftrightarrow\; \bigwedge_{q\in\mathsf{S}} K_p\Big(K_q\Phi_i \;\vee\; \Diamond\textsc{crash}(q,i)\Big)\ .$$

In the next section we show that knowing a decision determinant is obsolete requires a perfect failure detector.

5 Solving RDC

We prove that even if Consensus (or many other distributed coordination problems) can be solved, RDC cannot be. To do this, we first prove that when the crash-persistent storage is full, and one decision determinant must be purged, every process must be precisely accounted for regarding its knowledge of the decision determinant that is about to be purged, or its failure. We then prove that only a *perfect* failure detector (not even an eventually perfect failure detector) can give the necessary accurate information about process failures.

5.1 Knowing OBSOLETE(Φ_i)

Proposition 3 *Even if a process p knows a decision determinant, Φ_i, is obsolete, it cannot take any action that will make Φ_i not crash-persistent until it can account for every other process q either knowing Φ_i or being crashed,*

$$\begin{aligned}&K_p\textsc{obsolete}(\Phi_i) \Rightarrow\\ &\qquad K_p\textsc{c-persist}(\Phi_i) \;\vee\; \bigwedge_{q\in\mathsf{S}}\Big(K_pK_q\Phi_i \;\vee\; K_p\textsc{crashed}(q,i)\Big)\ .\end{aligned}$$

Proof By contradiction. If p knows Φ_i is obsolete, but does not know Φ_i is crash-persistent then p can imagine Φ_i will be purged from S if p next crashes or erases its local copy of Φ_i. If p does not know that q was aware of Φ_i, there is a p-equivalent (*i.e.*, $\sim_p$) cut on which q has never known about Φ_i. Finally, there is also a p-equivalent cut on which q is not crashed. Because purging a decision determinant neither causes processes to crash nor forces them to learn it, nothing prevents the worst case, where the conjunction, $\textsc{purged}(\Phi_i) \wedge \Box\neg K_q\Phi_i \wedge \Box\neg\textsc{crashed}(q)$, holds. $\boxed{\text{QED}}$

Proposition 4 *[à la Chandy-Misra [6]] Knowledge of decision determinants is local to a process,*

$$K_p K_q \Phi_i \Leftrightarrow \textsc{recv}^*_p(q, \Phi_i) \ .$$

Proof This follows from channel properties (p cannot be sure any particular message informing q of Φ_i will reach q) and because the event whereby a process learns information is not visible to remote processes. $\boxed{\text{QED}}$

Proposition 3 and Proposition 4 mean that a process must query all members of S regarding their knowledge of Φ_i before it can know $\textsc{obsolete}(\Phi_i)$ to overwrite a local copy of it. Furthermore, a process cannot overwrite its local copy until it knows any unresponsive process is, indeed, crashed.

Proposition 5 *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 could implement perfect failure detectors. Let $\mathcal{W}$ be the class of weak failure detectors of Chandra and Toueg [4]. Any member of $\mathcal{W}$ satisfies weak accuracy (some correct process is never suspected) and weak completeness (each crashed process is eventually permanently suspected by some correct process). 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. We can now construct a perfect failure detector from any member of the class $\mathcal{W}^+$.

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

Now take K as the suspicion list for a perfect failure detector; only crashed processes are ever in K (strong accuracy) and eventually every crashed process is in K (strong completeness). $\boxed{\text{QED}}$

5.2 Perfect Failure Detectors and RDC

Proposition 6 *In the absence of infinite storage capacity, unreliable failure detectors cannot be used to solve RDC if even one process can fail.*

PROOF Any solution to RDC must purge decision determinants if more than δ decisions are to be made, and only obsolete determinants can be purged safely (Proposition 1). Decision determinants are made crash-persistent (to f crashes) by storing them at (at least $f+1$) different sites; whenever a process overwrites its local copy of a crash-persistent decision determinant, it willfully renders the decision determinant not crash-persistent. To ensure safety, the process must know the decision determinant is obsolete. Now, Proposition 3 gives

$$K_p\text{OBSOLETE}(\Phi_i) \;\wedge\; \neg K_p\text{C-PERSIST}(\Phi_i) \Rightarrow K_pK_q\Phi_i \;\vee\; K_p\text{CRASHED}(q) \;,$$

which Proposition 5 proves is impossible for unreliable failure detectors. $\boxed{\text{QED}}$

Proposition 7 *In the absence of infinite storage capacity, eventually reliable failure detectors cannot be used to solve RDC if even one process can fail.*

PROOF The proof is similar to that of Proposition 6. Eventually reliable failure detectors do not give true knowledge of process crashes because the instant at which they become reliable, while guaranteed to occur, cannot be distinguished. A process must always consider that its failure detector is not yet reliable. $\boxed{\text{QED}}$

Proposition 8 *In the absence of infinite storage capacity, perfect failure detectors are necessary and sufficient for solving RDC if even one process can fail.*

PROOF For sufficiency, the algorithm below uses a perfect failure detector to purge obsolete decision determinants.

/* p wants to overwrite Φ_i. */
forall $q \in \mathsf{S} - \mathsf{K}$ {
    $send_p(q, RequestPurge(\Phi_i))$;
    await $K_p\text{CRASHED}(q)$ or $recv_p(q, K_q\Phi_i)$;
}
forall $q \in \mathsf{S} - \mathsf{K}$ $send_p(q, DoPurge(\Phi_i))$;
erase local copy of Φ_i.

For necessity, Strong Accuracy guarantees that suspicions are never erroneous. Thus, if p suspects q, then $K_p\text{CRASHED}(q)$, which Proposition 3 shows in necessary before purging. Until p suspects q, it must continue to wait for a message from q indicating $K_q\Phi_i$ (Proposition 4). Without Strong Completeness, if q is crashed, p may wait forever. $\boxed{\text{QED}}$

5.3 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 Distributed Consensus. The need for perfect failure detectors is attributable to the need to reach infinitely many decisions, the need to maintain them in crash-persistent storage for all processes that never crash, and the reality of finite storage space. We define *Generic Repeated Coordination* (GRC) in Definition 6, showing also that (repeated) Reliable Broadcast is an instance of GRC. In consequence, Proposition 9 proves any problem that uses Reliable Broadcast requires a perfect failure detector.

GRC differs from distributed coordination that is not repeated only in the clauses we call Unboundedness and Transience. Unpredictability means actions that arise in a run are not predetermined. In Distributed Conensus, for example, it means the initial condition values are randomly set; in GRC these unpredictable initial conditions exist for each action. Transience puts the onus on S to store information about actions that have arisen; the factors responsible for generating any action do not recur. Integrity prevents processes from simply taking every action in the name space. In Atomic Commit, it prevents processes from always deciding "abort". Agreement can be Uniform or not, and also includes the traditional Termination statement. Validity precludes solutions in which no actions are taken (and System Property 1 precludes solutions in which all processes crash themselves purposely). Unboundedness allows for infinite runs, and means no amount of storage obtained in advance will suffice for every run.

Proposition 9 *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 $|\mathsf{S}| > f \geq 1$ processes can crash and that crash-persistent storage can keep only δ actions simultaneously. Transience means information about actions must be made crash-persistent to ensure all correct processes have acces to it to satisfy Agreement. Unboundedness and Validity require an unbounded number of decision determinants to be kept in crash-persistent storage. Generalizing Proposition 2, at least $(i - \delta)$ must be obsolete and purged when the i^{th} action is taken. Only the Agreement condition can define OBSOLETE(), and it exempts only crashed processes from having to take actions. Proposition 3 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 stored. 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; for example, Distributed Consensus with a $\diamond\mathcal{W}$ failure detector requires a majority of S to implement crash-persistent storage, but with a $\mathcal{S}$ failure detector requires only two. In this way, the power of an adversary to crash processes is determined by the underlying distributed coordination problem, not that fact that the problem is

Definition 6

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 $do_p(a)$ 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.[a]

Transience Knowledge of an action can be permanently lost within S if it is not crash-persistent.

Integrity Only actions that arise in a run can be taken by the members of S.

Agreement If any (correct) process p in S executes the event $do_p(a)$ then every other correct process q in S inevitably executes $do_q(a)$.

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

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

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

Repeated Reliable Broadcast

Given the obligation set, S, let M be an infinite set of all possible messages, that processes deliver with the special event $deliver_p(m)$ for $p \in \mathsf{S}, m \in M$.

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

Transience Knowledge of a message can be permanently lost within S if it is not crash-persistent.

Integrity Only messages that are broadcast in a run can be delivered by the members of S.

Agreement If any (correct) process p in S executes the event $deliver_p(m)$, then every other correct process q in S inevitably executes $deliver_q(m)$.

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.

being solved repeatedly. Proposition 9 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.

In light of Propositions 6 and 9, the algorithm given by Chandra and Toueg [4] to solve Atomic Broadcast should be discussed. In essence, their solution assumes infinite capacity channels in using Repeated Reliable Broadcast to disseminate decisions on delivery order for Atomic Broadcast. If the underlying reliable channels have infinite capacity then no process ever blocks sending a reliable broadcast. However, if channels have finite capacity then a process needs recourse to some mechanism that releases it from having to send messages along that channel. As Proposition 6 shows, unreliable failure detectors cannot suffice. The Chandra and Toueg algorithm uses reliable broadcast to ensure all correct processes receive all information; unless a perfect failure detector is used, it is possible that a correct process will not receive all information.

Finally, as Proposition 9 depends solely on accurate failure dectection and infinite storage, our results are not sensitive to how processes communicate with each other; in particular, they hold for any *shared-memory* system with (crash-persistent) finite storage.

5.4 Consensus and Knowledge of Process Failures

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 folk-lore is misattributed. 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.

This can be explained by the fact that the Weak Accuracy property of unreliable failure detectors already encodes a primitive consensus construct. Specifically, there is consensus among all processes not to suspect (at least) one correct process. This, of course, is no guarantee that a suspected process is crashed. However, this hidden consensus structure can be used to bootstrap the consensus decision. In essence, the Chandra and Toueg solution discovers the identity of the highest-ranked process that has never (yet) been suspected. If the protocol terminates before this process is suspected, the value it proposes is adopted. Thus, (eventual) Weak Accuracy not only guarantees Liveness (such a process exists), but also provides a primitive consensus construct on which to build the solution.

6 Discussion and Future Work

We have shown that Reliable Broadcast and any other coordination problem that exempts an obliged process from coordinating only if it crashes, and that must be solved repeatedly requires either a perfect failure detector or the ability to store infinitely many of the coordinated actions. This work points to a number of important issues for any repeated coordination problem (Consensus or otherwise) that now bear further investigation.

6.1 The Need to Add Processes

In GRC, the correct members of S must take infinitely many actions. For any given coordination problem being considered, there is an inherent fault tolerance, f, limiting the number of crashes in S. To achieve infinitely many decisions then, at least some processes never, ever crash.[5] Outside theoretical models, this is unrealistic. On the other hand, if we admit that every process in S may eventually crash, there is no hope for infinite executions; one must be allowed to *add* processes to S. If that is the case, S becomes indexical; the composition is a function of time.

The notion of obligation must then be reconsidered. A process joining (the now indexical) $\mathsf{S}()$ late in the execution cannot be exepcted to take actions for which information has already been purged, though it may be required to take actions whose determinants are in crash-persistent storage when it joins $\mathsf{S}()$. Indeed, it is clear from the Agreement and Unboundedness conditions stated earlier, and in Formula 1, that whenever $\mathsf{S}()$ changes, the very problem itself changes.

This then raises questions about the power of an 'adversary' to crash processes; specifically, the fault-tolerance of GRC will be a function of an evolving $\mathsf{S}()$. This is not surprising, and should be seen as the logical extension of the usual case in which fault tolerance is a function of a static, initial S. The power of the adversary is no longer limited in the number of processes it can crash over a run, but in the spacing of these crashes.

6.2 The Need to Remove Processes

Lacking perfect failure detectors, the need to know $\textsc{crashed}(q)$ should be replaced by something a process in $\mathsf{S}()$ can actually learn. For most applications' consistency, it should also be something that is indistinguishable from q having crashed (certainly to clients of any service $\mathsf{S}()$ is implementing). There has been some work considering GRC with broader exemptions that are indistinguishable from $crash_q$ [18, 20]. Sabel and Marzullo have taken a similar approach in simulating failure detectors [21]. Observe that $crash_q$ affects GRC by dividing S into one subset of processes that are still obliged to coordinate, and one subset that is exempt from coordinating. Processes that crash and recover before any desicion determinant has been purged are not problematic; processes that recover after a decision determinant has been purged must nonetheless (for safety) be prevented from making inconsistent decisions.[6] If one considers similarly the subset of $\mathsf{S}()$ that, at any point in time, may possibly make another decision (those processes that have neither crashed nor missed any purged decision determinants), then $crash_q$ removes q from this subset, as does missing a decision determinant that has been purged. This perspective suggests that faultiness is as much an issue of (storage) space as it is of time, and that, until perfect failure detectors are realizable, instances of GRC should consider a truly dynamic obligation set that adds and removes members.

Finally, System Property 1 disallows solutions in which a process 'crashes itself' when it realizes

[5] We discuss the difficulties posed by processes recovering shortly.

[6] They can recover as 'new' processes and join $\mathsf{S}()$ as described above.

it cannot continue safely. This could easily be the case in sequential GRC where a process can maintain a counter with the index of the last decision it made. Suppose p has made three decisions. If, in attempting to learn the value of the fourth, p is informed the earliest decision on record is the ninth (p missed intermediate decisions due to asynchrony or network problems), then p knows it cannot continue. As p is no longer useful to the other members of $\mathsf{S}()$ it should voluntarily leave $\mathsf{S}()$, then ask to be readmitted.

6.3 Persistence and Membership

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

Persistence The solution ensures that all relevant information will available to any process (and that the information will be used by the process).

Membership The solution does not allow any process to decide unless every other process either has the decision information or is known to never make another decision (whether it crashes or not).

We have shown all solutions embody a combination of the two. When there is no storage space, Persistence is Membership. Conversely, Persistence has no element of Membership only if storage space is infinite.

If the specification requires that any process that never crashes must reach a decision state, then only Persistence is acceptable. If the specification only requires that inconsistent decisions never be made (intrinsically allowing some processes to never decide) then both are acceptable. If there is no guarantee that decision choices can be made eternally persistent, then Membership is necessary when information is purged.

Acknowledgements

I thank Rachid Guerraoui and André Schiper at EPFL for their many helpful discussions and comments on this work.

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 and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Tenth PODC*, pages 325–340. ACM, 1991.

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

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

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

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

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

[10] R. Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus Problems. In *9th WDAG. Springer Verlag Lecture Notes in Computer Science*, page to appear, 1995.

[11] V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations.* PhD thesis, Harvard University, 1984. Available as TR-11-84.

[12] 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.

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

[14] 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.

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

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

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

[18] 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).

[19] G. Neiger and S. Toueg. Automatically Increasing the Fault-tolerance of Distributed Algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[20] A. Ricciardi. Dissecting Distributed Coordination. In *9th WDAG. Springer Verlag Lecture Notes in Computer Science*, page to appear, 1995. Univ Texas TR ECE-PDS-9502.

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

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