Dissecting Distributed Coordination*

Aleta Ricciardi[1]

The University of Texas at Austin, Austin TX 78712, USA

Abstract. This paper derives necessary and sufficient communication for distributed applications that perform certain actions *uniformly* in asynchronous systems. We show there is an essential structure of information flow in any solution to Uniform Coordination, suggesting message-minimal solutions. We show it is necessary for processes to conspire against each other to make progress, and we show this conspiracy requires processes to stop communicating with each other. This, we show, renders Uniform Coordination insensitive to channel delivery guarantees. We introduce the notion of *exempting* processes from coordinating. We show that 'primary partition' behavior (Isis) arises from the desire to make exempt an process indistinguishable from a crashed process. Defining weaker exemptions for distributed coordination problems gives rise to many problems solvable in asynchronous systems as well as in systems that partition. **Keywords:** Agreement, asynchronous, exemption, failures, uniformity.

1 Introduction

Distributed computing systems can broadly be viewed as alternating periods of coordination and autonomy. Coordination can occur intermittently or more frequently, and each instance will have different requirements regarding when the coordinated activity must complete, which processes must participate, and conditions under which some may later be deemed exempt. Research has tended to focus on Distributed Consensus, a simple yet strict coordination problem, and the inability of attaining it in asynchronous systems [7]. Recent work has found ways around the impossibility result: Chandra, Hadzilacos and Toueg augment the asynchronous environment with failure detectors, giving minimal conditions these must satisfy to assist a group of processes solve Consensus [3], while randomized techniques (*e.g.*, [6]) give probabilistic solutions. This paper and others (*e.g.*, [12, 15]) consider weaker coordination problems that are, without external assistance, solvable in asynchronous environments.

This work puts forth the notion that very useful coordination can be achieved with minor changes to provably unsolvable problems (*e.g.*, Consensus, Atomic Broadcast). Specifically, we view the requirement that all non-crashed[2] processes

* This research is supported by the NSF under grant number ASC-9318151, and a University Research Institute Summer Research Award from the Univerisity of Texas.

[2] We adopt the view (proposed in [7]) that any interesting algorithm would have all non-crashed processes decide.

reach a consistent decision as a direct way of simply *exempting* crashed processes from having to make a decision. Certainly, a coordination problem must exempt crashed processes from acting, but requiring that all non-crashed processes must act is perhaps too severe. It has been shown that systems can reach agreement with some non-crashed processes not deciding, and that one can make these executions indistinguishable from those in which the non-deciding processes do crash [15, 16].

To achieve indistinguishability, processes that are not known to decide and do not crash must be forced to behave as if they were crashed – they must (at least) be prevented from making further decisions. This induces a 'primary partition', a unique subset of processes that continues making decisions and prevents others from doing so. We show that a set of processes can effectively crash a member only by refusing to communicate with it. By being *shunned* in this way, a process is denied information essential to it making further decisions. The need for this information arises from the safety condition and the need to exact control over processes. Using results from Chandy and Misra [5], we derive the essential communication structure for any solution, then use this to determine which information must be kept from a process to disable it. We show primary partition behavior (implemented in systems like Isis [2]) is directly attributable to any exemption that requires a process to cease taking actions if it is exempt from taking any action.

Thinking of distributed coordination in terms of behavior that should be prevented if a process does not participate in a given coordinated activity is new and may lead to more appropriate, balanced coordination problems. For example, broader notions of exemption accurately describe coordination problems that can make progress in multiple sides of a partition. This is especially important for computations that occur over wide-area networks, which are (now and for the foreseeable future) prone to partition. Even in LANs, Transis [1] and Horus [17] permit some progress during a partition; for efficiency and correctness it is important to know which actions need not be resolved at merge time.

2 Asynchronous System Model

The system consists of an infinite domain of process identifiers, Proc, though at any point in time only a finite subset of them are executing. The system also has a network of channels between processes. Processes communicate with each other only by passing messages through the network. The system is asynchronous in that there is no common global clock, and messages may be arbitrarily long in transit. A process p fails by crashing, but otherwise follows its assigned protocol. While messages between processes may be lost (though not corrupted), every message sent has some non-zero probability of being received. If two processes p and q both never fail, every message from p will eventually reach q. In this environment no process can determine whether a process is unresponsive because it is crashed or because of transient performance factors.

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) 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 $\mathsf{S} \subset \mathsf{Proc}$ is a tuple of finite process histories, one for each $p \in \mathsf{S}$. We assume inter-event causality [10] and consistent cuts [4] are understood.

A *system run* is a tuple of infinite process histories, one for each process that ever executes.[3] For each run of the system, the initial system composition S_0 is the set of all processes whose *start* events are not causally related; the initial cut consists only of all the $start_p$ events for $p \in \mathsf{S}_0$.

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 over S and c' over S', $c \leq c'$ if and only if $\mathsf{S} \cap \mathsf{S}' \neq \emptyset$ and, for any process in the intersection, its history component in c is a 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.

2.1 Distributed, Coordinated Actions

As in [12], we are concerned with applications in which specific events must occur at all processes if at any, and designate a special set of events, called *actions*. While the set of actions can be described (*e.g.*, withdrawals from accounts at the Black Hole Savings and Loan) the specific actions to appear in any execution are not known *a priori* (*e.g.*, from which account, how much, and when). An action will arise unpredictably at a single process in any given run. If α arises at p in run ρ we say p *owns* α in ρ; p_α denotes the owner of α. The particular event whereby a process learns about the existence of an action does not have any externally observable effect on that process.

2.2 The Formal Language

We use a logic supporting temporal and knowledge modalities to define the problem. Formulas are evaluated on consistent cuts. 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 - branching time) holds on c if and only if φ holds on *some* $c \leq c'$, in *every* run that includes c. $\Box$ and $\Diamond$ are not duals. $\diamondminus\varphi$ (in the past) holds on c if φ holds on some $c' \leq c$.

Knowledge. [9] $K_p\varphi$ (p knows) holds on c if and only if φ holds on all cuts $c' \sim_p c$. The event $learn_p(\varphi)$ refers to the place in p's history on the earliest cut satisfying $K_p\varphi$.

Crashed processes cannot 'know' anything; that is $K_p\varphi \Rightarrow \neg\textsc{crashed}(p)$. $learn_p(\varphi)$ can easily be affixed with an index if $K_p\varphi$ is not stable, though this will not be necessary in our discussions.

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

3 Uniform Distributed Coordination

Loosely stated, Uniform Agreement [8] requires all processes to reach agreement on a value, if any process decides on that value; non-uniform agreement only requires consistent behavior from the processes that never crash. Uniformity is more appropriate for our concerns. In asynchronous systems that experience only crash failures, up to the point a process crashes it behaves 'correctly' (Gopal and Toueg point out, too, that non-uniform agreement originally considered malicious failures [11]). For a participating process, deciding whether to ignore the actions of processes that will later be deemed incorrect is akin to predicting the future. As well, some applications require that the external world never be able to discern any inconsistency among the processes participating in the agreement protocol. Whenever a group of processes cooperate to implement a service that must appear as if it were being run by a single, fault-tolerant process (*e.g.*, key distribution services, controllers), the actions of *all* processes will be important, not just the ones that never crash. Non-uniform agreement exempts processes that will later be deemed faulty from behaving consistently, even before they encounter a fault.

We use the following formulas to define Uniform Coordination; they are all stable (*i.e.*, $\varphi \Rightarrow \Box\varphi$). $\text{SEND}_p(q, m)$ holds on c if and only if $send_p(q, m)$ is an event in c. $\text{SEND}^*_p(q, m)$ holds when there is a sequence of processes $p = p_0, ..., p_k = q$ and messages, $m_0, ..., m_{k-1} = m$ such that $\text{SEND}_{p_i}(p_{i+1}, m_i)$ holds on c and (the contents of) each m_1 implies (the contents of) m_0. We say m *acknowledges* m_0 and use $\text{ack}(m_0)$ for m to make this explicit. Similarly define $\text{RECV}_q(p, m)$ and $\text{RECV}^*_q(p, m)$. $\text{CRASHED}(p)$ holds on c if $crash_p$ is an event in c. $\text{BEFORE}(e_1, e_2)$ holds on c if $e_1 \rightarrow e_2$ in c. $\text{DID}_p(\alpha)$ holds on c if $do_p(\alpha)$ is an event in p's history component of c. 'α' holds on c if some non-crashed process knows the action α has arisen.

For the problems we consider, actions to be taken uniformly arise unpredictably in that processes receive stimuli from the external world. Because processes may crash and the external stimuli alerting a process to an action may occur only once, the existence of an action is tied to it being known by a (non-crashed) process. We associate an action's initial existence with its owner's knowledge of it, 'α' $\Rightarrow \diamondminus K_{p_\alpha}$'$\alpha$', and state that p_α is necessarily the first process to learn 'α' : $\text{BEFORE}(learn_{p_\alpha}(\text{'}\alpha\text{'}), learn_q(\text{'}\alpha\text{'}))$ for all other q.

3.1 Process Exemptions

The nature of asynchronous environments leads to a number of necessary components to *any* solution to Uniform Coordination. We must consider these before giving a complete formal specification for the problem. The non-triviality condition we will put forth forces each process to attempt to execute all actions it knows about, and the safety condition will force these actions to be done uniformly. In so doing, the safety condition will explicitly quantify a set of processes *obliged* to do each action, as well as conditions under which some of these processes may later be deemed exempt. An obvious exemption is the crash of an obliged process, but asynchrony necessitates others.

The usual statement for many distributed coordination problems requires all 'correct' processes to take the same actions, which is one way of exempting crashed processes. While crashed processes must be exempt, this particular statement is problematic. For example, the need to distinguish crashed from non-crashed processes is often cited as the reason behind Consensus and Atomic Broadcast. As well, a solution must store all values that have been decided. Each value must be store indefinitely, so that a non-crashed process, no matter how slow, can retrieve it, and it must stored at multiple sites to persist despite process crashes.

The Uniform Coordination problem described here and in [12], and the Group Membership Problem described in [15] broaden the notion of exemptions so that solutions are not required to determine the set of crashed processes exactly, or save actions indefinitely. In theory, an exemption could be granted for any reason, though we focus on process crashes and 'apparent crashes'. An exemption can limit the future behavior of a process : it can be terminal (once a process is declared exempt from taking any action, it can never take another action), restrictive (a process can only take actions from some subset), or conditional (a process is unable to take actions until some other condition is met).

Uniform Coordination, as well as the coordination achieved in Isis, produce runs that are indistinguishable from those in which the exempt processes actually crashed; that is, we are interested in a terminal exemption. Terminal exemptions allow one to reach meaningful agreement in asynchronous systems by not equating exemption with accurate failure determination.

Definition 1. A process p is *disabled*, and the formula $\text{DIS'D}(p)$ holds, exactly when p will never take any action it has not already taken,

$$\text{DIS'D}(p) \stackrel{\text{def}}{=} \forall\alpha : \Big(\neg\text{DID}_p(\alpha) \Rightarrow \Box\neg\text{DID}_p(\alpha)\Big) \ . \qquad (1)$$

It follows that a crashed process is disabled.

3.2 Permission for Actions

Assume process p wishes to perform action α and let S_α be the set obliged to do α along with p (Section 3.4 discusses this). Since α is initially known only to p_α, the action should be *announced* to the obligation set, S_α, for no member of S_α can take an action it does not know about. For now let $Annc_p(\mathsf{S}_\alpha)$ be a loose way of p propagating knowledge of α (*e.g.*, multi-cast that is not failure atomic). Intuitively Algorithm 1 (Figure 1) is not safe because p cannot guarantee it will not crash between doing α and announcing it.

To effect a terminal exemption, the system must have some means of making exempt processes appear crashed. In the context of Uniform Coordination, that means exempt processes may take only finitely many actions. The notion of *permission* to take an action arises from the need to make some processes appear crashed; that is, terminally exempt processes must have their ability to take new actions revoked. In the absence of an omnipotent external entity, granting permission and revoking it falls to the members of S_α. In this way Algorithm 2 is not safe for two reasons: first there is no way of preventing p from ever taking actions, and second the network cannot guarantee any member of S_α will receive this particular announcement; should p fail immediately after taking α, knowledge of the action may be lost forever.[4]

/* Alg 1 */	/* Alg 2 */	/* Alg 3 */
do α; $Annc(\mathsf{S}_\alpha, \alpha)$.	$Annc(\mathsf{S}_\alpha, \alpha)$; do α.	$Annc(\mathsf{S}_\alpha, \alpha)$; when $K_p\textsc{permit}_p(\alpha)$ do α.

Figure 1. Possible Uniform Coordination Solutions.

Thus, while announcing an action seems necessary, it is not sufficient. A process must also receive some assurance before taking an action implying it will be taken uniformly; again, lacking some sort of oracle, this assurance ultimately comes from S_α. The formula $\textsc{permit}_p(\alpha)$ holds whenever it is 'safe' for p to take α. Section 4.1 derives minimal conditions for defining $\textsc{permit}_p(\alpha)$, and therefore safety. $\textsc{permit}_p(\alpha)$ and $\textsc{did}_p(\alpha)$ interact as expected, and liveness will prevent $\textsc{permit}_p(\alpha)$ from being identically false. We stipulate that initially no actions are permitted, that if a process takes an action, it has permission for it, and that actions known to be permitted are eventually taken, provided a process does not crash.

3.3 Specification

Elsewhere [14] we have argued that many important problems in distributed computing can be specified by variants on three clauses : Uniformity, Sequence, and Termination. Here, we focus on the Uniformity clause which is the basis of other coordination problems. Variations on the Uniformity clause will define the set of processes obliged to take an action, and the exemption, which for our purposes, is $\textsc{dis'd}()$. We are concerned with properties of all solutions to Uniform Coordination, and so confine the discussion to the set of system runs generated by them. Hereafter, when we prove or require $c \models \varphi$, it should be understood in the context only of these runs, not the set of all runs of an asynchronous system.

The clause specifying Uniformity is similar to that in [12], but refers to knowledge of permission rather than action. Since a process must eventually act if it knows it has permission and cannot act without permission, using $K_p\textsc{permit}_p(\alpha)$ is equivalent and more direct. The Uniformity condition requires that if one member of S_α gets permission to take an action, then all other

[4] It will be clear later (Theorem 16) that even if the network guarantees delivery, Algorithm 2 is not safe. The members of S_α may already have decided to 'shun' p so that p not waiting for permission violates safety. The need to shun arises the need to prevent exempt processes from taking future actions.

processes in that set are obliged to (try to) get permission for it also; the only processes exempt from taking uniform actions are those that crash or become otherwise disabled,

$$p \in \mathsf{S}_\alpha \wedge K_p\textsc{permit}_p(\alpha) \;\Rightarrow\; \bigwedge_{q\in \mathsf{S}_\alpha} \Diamond\Big(K_q\textsc{permit}_q(\alpha) \vee \textsc{dis'd}(q)\Big)\,. \tag{2}$$

Formula 2 does not require p to know which of the two clauses, $K_q\textsc{permit}_q(\alpha)$ or $\textsc{dis'd}(q)$, will hold before p has permission to take α. This allows runs of limited divergence (see Figure 2), in which processes with no intention of adhering to the uniformity constraint for actions initiated by other processes are, for a finite period, permitted to execute their own actions. While the actions these 'outlaw' processes take must, of course, be taken by all others that are never disabled, an outlaw process need not conform to any other process's behavior. A strong Termination clause (see [14]) would prevent such behavior.

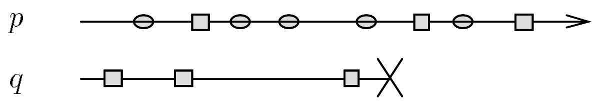

Figure 2. Process q displays finite divergence.

The non-triviality and liveness clauses state that knowledge of an action forces a process to try to take it, and preclude solutions in which all processes take a finite number of actions then quit.

3.4 Obligation Sets

A coordination problem will define obligation sets to reflect the desired degree of system-wide uniform coordination. For example, S_α can be static, or it can be dynamic [12]; it can be defined according to a single process (like p_α) or many processes. It should also be noted that obligation sets are closely related to the exemption. Our $\textsc{dis'd}()$ exemption is defined in terms of denying a process permission for all future actions. If the obligation set of a future action is not related to that of a pending action, drastic measures are needed to ensure $\textsc{dis'd}()$. Here, we consider a static set of processes that are mutually bound to execute the same set of actions. As Corollary 15 shows, there will always be a need to add processes; it is beyond the scope of this paper to deal with those additions, and how they affect the underlying coordination problem.

4 Results

As Propositions 2 and 3 are similar to well-known results by Chandy and Misra [5], we do not include their proofs here (see the full paper [13]).

Proposition 2. *Knowledge of actions is propagated by communication,*

$$K_p\,\text{`}\alpha\text{'} \Leftrightarrow \begin{cases} \neg\textsc{crashed}(p) & p = p_\alpha \\ \textsc{recv}^*_p(p_\alpha, \mathit{ack}(\text{`}\alpha\text{'})) \wedge \neg\textsc{crashed}(p) & p \neq p_\alpha \ . \end{cases}$$

We no longer explicitly state or prove the disclaimer that process knowledge depends on the process not having crashed. From Proposition 2, if p_α never sends a message mentioning 'α', then no process will ever know 'α', meaning that $\mathit{Anncp}_{p_\alpha}(\mathsf{S}, \alpha)$ involves p_α sending messages to each member of S. Thus, $\mathit{Anncp}_{p_\alpha}(\mathsf{S}, \alpha)$ is $\forall p \in \mathsf{S} : \mathit{send}^*_{p_\alpha}(p, \alpha)$, or more generally $\mathit{Mcast}_{p_\alpha}(\mathsf{S}, \alpha)$.

Proposition 3. *A process's knowledge of an action is itself a local formula,*[5]

$$K_p K_q\,\text{`}\alpha\text{'} \Leftrightarrow \textsc{recv}^*_p(q, \mathit{ack}(\text{`}\alpha\text{'})) \wedge \neg\textsc{crashed}(p) \ .$$

The next theorem states that before a process can take an action, it must be able to account for all other obliged processes (either a process knows about the action or will be disabled). This is stronger than the Uniformity statement itself which only requires p to know that eventually one of the two will hold, and is directly attributable to the asynchronous system (though see Footnote 4).

Theorem 4. *[Accountability Theorem] If process p knows it has permission to take action α, then either p knows $K_q\,\text{`}\alpha\text{'}$ or p knows $\Diamond\textsc{dis'd}(q)$ for every $q \in \mathsf{S}$,*

$$K_p\textsc{permit}_p(\alpha) \Rightarrow \bigwedge_{q \in \mathsf{S}} K_p K_q\,\text{`}\alpha\text{'} \vee K_p\Diamond\textsc{dis'd}(q) \ . \tag{3}$$

Proof. Formula 3 differs from the Uniformity statement in that not only must p know that eventually q will learn about α (Proposition 5), but that "eventually" is not good enough (Proposition 6); p must have definitive evidence that q already knows about α. Moreover, in this case, the knowledge operator distributes over the disjunction meaning that p must know one way or the other whether q will be disabled or already knows about the action (Proposition 7).

Proposition 5. *Before $p \in \mathsf{S}$ can know it has permission for α, it must know every $q \in \mathsf{S}$ will eventually know that α has arisen or will eventually be disabled,*

$$K_p\textsc{permit}_p(\alpha) \Rightarrow K_p\Diamond\Big(K_q\,\text{`}\alpha\text{'} \vee \textsc{dis'd}(q)\Big) \ .$$

Proof. Replacing $K_q\textsc{permit}_q(\alpha)$ with K_q'α' in Formula 2 is legitimate because $K_q(P \vee Q) \wedge (P \Rightarrow P_1) \Rightarrow K_q(P_1 \vee Q)$, and because $K_q\textsc{permit}_q(\alpha) \Rightarrow K_q$'$\alpha$'. $\Box_{Pr5}$

Proposition 6. *A process cannot know only that another process will eventually know about an action, $K_p\Diamond K_q\,\text{`}\alpha\text{'} \Rightarrow K_p K_q\,\text{`}\alpha\text{'}$.*

[5] The LHS should be $K_p\diamondminus K_q$'α' to capture the possibility that q may be crashed, though we avoid this formulation for readability.

Proof. The proof is by contradiction. Suppose $c \models K_p \Diamond K_q `\alpha' \wedge \neg K_p K_q `\alpha'$, and define $\mathsf{knows}(c, `\alpha') \stackrel{\text{def}}{=} \{r \in \mathsf{S} \mid c \models K_r `\alpha'\}$. Then p can imagine a $c' \geq c$ such that all channels are empty[6] on c' and

$$c' \models \bigwedge_{r \in \mathsf{knows}(c, `\alpha')} \text{CRASHED}(r) \wedge \neg\text{CRASHED}(q) \wedge \neg K_q `\alpha' \ .$$

After c', q can never learn 'α', so that c cannot satisfy $K_p \Diamond K_q `\alpha'$. $\Box_{Pr6}$

Proposition 7. *To know of a remote process's eventual knowledge of an action or its eventual demise, is to either know that it already knows about the action or to know that it will eventually be disabled,*

$$K_p \Diamond \Big(K_q `\alpha' \vee \text{DIS'D}(q) \Big) \Rightarrow K_p K_q `\alpha' \vee K_p \Diamond \text{DIS'D}(q) \ .$$

Proof. By way of contradiction, we construct a $\sim_p$-equivalent cut on which $\Diamond \Big(K_q `\alpha' \ \vee \ \text{DIS'D}(q) \Big)$ does not hold. Assume

$$c \models K_p \Diamond \Big(K_q `\alpha' \ \vee \ \text{DIS'D}(q) \Big) \ \wedge \neg K_p K_q `\alpha' \ \wedge \ \neg K_p \Diamond \text{DIS'D}(q) \ ,$$

and let $\mathsf{knows}(c, `\alpha')$ be as in Proposition 6. Construct $c_1$ from $c$ by crashing all processes (except $p$) in $\mathsf{knows}(c, `\alpha')$ and dropping any message currently in transit to q. Further impose that 'α' does not arise again in any completion of c_1. Let c_2 be identical to c_1 except that p crashes immediately next. That is, $c_2 \models \Box \neg K_q `\alpha'$ and so $c_1 \models \neg \Diamond K_q `\alpha'$. To satisfy the hypothesis, $\Diamond \text{DIS'D}(q) \wedge \neg K_q \Diamond \text{DIS'D}(q)$ must hold on c_1.

Lacking an omniscient, omnipotent entity, it must be that some subset of S jointly control $\text{DIS'D}(q)$ (and therefore $\Diamond\text{DIS'D}(q)$). By definition $\Diamond\text{DIS'D}(q)$ holds only if inevitably, q will never learn it has permission to take new actions; intuitively, these controlling processes somehow prevent q from learning it has permission. Since p does not know $\Diamond\text{DIS'D}(q)$, p can imagine the q-controlling processes have not yet decided to disable q. Since failures are independent, p can also imagine a completion of c_1 in which these (and any future) q-controlling processes crash before deciding to disable q and that q itself can learn they are disabled! Nothing now prevents q from learning it has permission for new actions, the ability to disable q is lost, and q will never learn 'α'. That is, $\exists c_1 \sim_p c$, such that $c_1 \models \neg \Diamond K_q `\alpha' \ \wedge \ \neg \Diamond \text{DIS'D}(q)$ if and only if $c_1 \models \neg \Big( \Diamond K_q `\alpha' \ \vee \ \Diamond \text{DIS'D}(q) \Big)$ implying $c_1 \models \neg \Diamond \Big(K_q `\alpha' \ \vee \ \text{DIS'D}(q) \Big)$.$\Box_{Pr7}$ and $\Box_{Th4}$

Proposition 8. *If process p knows it has permission to take action α, then p received a message implying $K_q `\alpha'$, or p knows $\Diamond\text{DIS'D}(q)$, for $q \in \mathsf{S}$,*

$$K_p \text{PERMIT}_p(\alpha) \Rightarrow \neg\text{CRASHED}(p) \ \wedge \bigwedge_{q \in \mathsf{S}} \text{RECV}^*_p(q, ack(`\alpha')) \vee K_p \Diamond \text{DIS'D}(q) \ .$$

Proof. Apply Proposition 3 to Theorem 4. $\Box$

[6] The proposition holds even when channels guarantee delivery. (Theorem 16).

4.1 Permission Revisited

At this point, a very simple definition of permission is evident. Given what must be true for a process to know it has permission, we can reason conversely and describe the condition that immediately causes a process to know it has permission. Theorem 4 and Proposition 8 mean that knowledge of permission is tied to a process receiving a message from each of the members of S that the requester does not know will eventually be disabled. That is, we can strip the K_p knowledge operators from Formula 3 to obtain

$$\textsc{permit}_p(\alpha) \Rightarrow \bigwedge_{q \in \mathsf{S}} K_q\text{‘}\alpha\text{’} \vee \Diamond\textsc{dis'd}(q) \ .$$

The simplest, non-trivial definition of permission makes this an equivalence.

Definition 9. Permission for p to take α holds exactly when ever member of S has either learned about α or will inevitably be disabled,

$$\textsc{permit}_p(\alpha) \stackrel{\text{def}}{=} \bigwedge_{q \in \mathsf{S}} K_q\text{‘}\alpha\text{’} \vee \Diamond\textsc{dis'd}(q) \ .$$

As per Section 3.2, initially no actions are permitted because no actions are initially known, and because processes do not immediately crash themselves. While this definition does not discriminate between members of S, specific applications would likely restrict permission depending on, say, a process's role within the obligation set. $\textsc{permit}_p(\alpha)$ can hold even after p has crashed or been disabled, though if p is crashed it cannot act. When p is disabled but not crashed, knowledge of $\textsc{permit}_p(\alpha)$ must be kept from p, and the next section derives necessary conditions to ensure this.

4.2 Disabling a Process

In this section, we derive the necessary conditions for preventing a process from learning it may have permission, thereby disabling the process. Proposition 8 and the definition of permission mean that if q is disabled (yet not crashed) there is some member of S for which q can never account. Proposition 10 instantiates $\textsc{permit}_p(\alpha)$ to make Proposition 8 an equivalence, and Theorem 11 shows that a process is disabled if it can never account for one of the other participants.

Proposition 10.

$$K_p\textsc{permit}_p(\alpha) \ \Leftrightarrow \ \neg\textsc{crashed}(p) \ \wedge \Big(\bigwedge_{q \in \mathsf{S}} \textsc{recv}^*_p(q, ack(\text{‘}\alpha\text{’})) \vee K_p\Diamond\textsc{dis'd}(q)\Big).$$

Proof. For the "$\Leftarrow$" direction, apply Proposition 3 and propositional logic. $\square$

Theorem 11. *If $p \in \mathsf{S}$ is not crashed, then it is disabled if and only if for every action it may ever be required to take, there is some $q \in \mathsf{S}$ for which p can never account,*

$$\begin{aligned} & p \in \mathsf{S} \wedge \neg\textsc{crashed}(p) \Rightarrow \\ & \Big(\textsc{dis'd}(p) \;\Leftrightarrow\; \forall\alpha : \Big(\Diamond K_p\text{`}\alpha\text{'} \wedge \neg K_p\textsc{permit}_p(\alpha) \;\Rightarrow \\ & \qquad \bigvee_{q\in\mathsf{S}} \Box\neg\textsc{recv}^*_p(q, ack(\text{`}\alpha\text{'})) \wedge \Box\neg K_p\Diamond\textsc{dis'd}(q)\Big)\Big) \;. \end{aligned} \tag{4}$$

Proof. Let $A_\alpha \stackrel{\text{def}}{=} K_p\textsc{permit}_p(\alpha)$, and let

$$B_\alpha \stackrel{\text{def}}{=} \bigwedge_{q\in\mathsf{S}} \textsc{recv}^*_p(q, \mathrm{ack}(\text{`}\alpha\text{'})) \vee K_p\Diamond\textsc{dis'd}(q) \;.$$

Proposition 8 says $A_\alpha \Rightarrow B_\alpha$, and Proposition 10 says $A_\alpha \Leftrightarrow \neg\textsc{crashed}(p) \wedge B_\alpha$.
"$\Rightarrow$" Assuming $\textsc{dis'd}(p)$ and $\neg\textsc{crashed}(p)$, let α be any action such that $\Diamond K_p\text{`}\alpha\text{'} \wedge \neg A_\alpha$.

a) Because p cannot be prevented from taking α if A_α holds, $\textsc{dis'd}(p) \wedge \neg A_\alpha \Rightarrow \Box\neg A_\alpha$.

b) Manipulating Proposition 10, means $\neg A_\alpha \Leftrightarrow \neg B_\alpha \vee \textsc{crashed}(p)$ is valid, as is $\neg A_\alpha \wedge \neg\textsc{crashed}(p) \Rightarrow \neg B_\alpha$, since the other disjunct cannot hold. Then $\Box$-generalization gives

$$\Box\neg A_\alpha \wedge \Box\neg\textsc{crashed}(p) \Rightarrow \Box\neg B_\alpha \;.$$

c) Let $\mathsf{awol}_p(c, \alpha)$ be the subset of S that p cannot account for regarding α at cut c. Expanding B_α gives

$$\Box(\neg B_\alpha) \;\Leftrightarrow\; \Box\Big(\bigvee_{q\in\mathsf{S}} \neg\textsc{recv}^*_p(q, \mathrm{ack}(\alpha)) \wedge \neg K_p\Diamond\textsc{dis'd}(q)\Big) \;,$$

and so $\Box\neg B_\alpha$ holds exactly when $|\mathsf{awol}_p(c', \alpha)| \geq 1$ for $c' \geq c$. As processes are never added to $\mathsf{awol}_p(c, \alpha)$, there must be some q that is in every $\mathsf{awol}_p(c', \alpha)$,

$$\Box\neg B_\alpha \;\Leftrightarrow\; \bigvee_{q\in\mathsf{S}} \Box\neg\textsc{recv}^*_p(a, \mathrm{ack}(\text{`}\alpha\text{'})) \wedge \Box\neg K_p\Diamond\textsc{dis'd}(q) \;.$$

d) Finally, whether p eventually crashes is irrelevant if p is already disabled: $\Box\neg A_\alpha \wedge \neg\textsc{crashed}(p) \Rightarrow \Box\neg B_\alpha$. To see this, suppose, by way of contradiction, that if p eventually crashes, B_α will eventually hold. Proposition 10 says if B_α ever holds then so does A_α, contradicting $\Box\neg A_\alpha$.

"$\Leftarrow$" Assume $\neg A_\alpha \wedge \neg\textsc{crashed}(p) \wedge \Box\neg B_\alpha$, for all α that p will learn about. Proposition 10 is equivalent to $\neg B_\alpha \vee \textsc{crashed}(p) \Leftrightarrow \neg A_\alpha$. $\Box$-generalization then gives $\Box(\neg B_\alpha \vee \textsc{crashed}(p)) \Leftrightarrow \Box\neg A_\alpha$. By hypothesis, $\Box\neg B_\alpha$ holds, and by introduction, so does $\Box(\neg B_\alpha \vee \varphi)$ for any formula φ, in particular, $\neg\textsc{crashed}(p)$. Since $\Box\neg A_\alpha$ holds, so does $\Box\neg\textsc{did}_p(\alpha)$, and therefore $\textsc{dis'd}(p)$. $\Box$

Theorem 11 implies that q, acting alone, can disable p by guaranteeing p never receives messages (from q) acknowledging actions p is trying to take; in Formula 4 we use $\Box\neg\text{SEND}^*_q(p, \text{ack}(+))$ to reflect q's intentional decision to disable p by ceasing to send acknowledgments to p. This is not enough, for any system can then partition so that all subsets ignore the others giving no guarantee any are disabled.

$$\begin{aligned}&\text{DIS'D}(p) \;\Leftrightarrow\; \exists q \in \mathsf{S} : \Box\neg\text{SEND}^*_q(p, \text{ack}(+)) \;\wedge\; \Box\neg K_p \Diamond\text{DIS'D}(q) \;\Leftrightarrow\\ &\exists q \in \mathsf{S} : \Box\neg\text{SEND}^*_q(p, \text{ack}(+)) \;\wedge\\ &\qquad\qquad \Box\neg K_p\Big(\exists r \in \mathsf{S} : \Box\neg\text{SEND}^*_r(q, \text{ack}(+)) \;\wedge\; \Box\neg K_q \Diamond\text{DIS'D}(r)\Big).\end{aligned}$$

Not only should p never receive future acknowledgments from q, but p should also never learn of a successful effort by any other process(es) to disable q. In a system consisting of only p and q, if p really is disabled, the system is asymmetric (q can disable p but not vice versa). A symmetric system of two processes cannot progress if either tries to disable the other; in larger systems this implies a coordinated conspiracy to keep information from p. The most stringent conspiracy would involve $(n-1)$ processes (for $n > 2$), but a *majority* suffices.[7] For Uniform Coordination, majorities determine which set of processes is not disabled; that is, which subset form the primary partition.

We say *q is shunning p* once q has intentionally ceased sending messages of any sort to p,[8] and denote it by $\text{SHUN}_q(p)$. As $\text{SHUN}_q(p)$ and $\text{CRASHED}(q)$ achieve the same end regarding p, and to p they are indistinguishable, we obtain,

$$\text{SHUN}_q(p) \vee \text{CRASHED}(q) \;\Leftrightarrow\; \Box\neg\text{SEND}^*_q(p, +)\ .$$

The next propositions prove that q can disable p if and only if it actively conspires with a majority of p's (*i.e.*, their) cohorts to shun p, and that q only learns this by communicating with this majority. Let M be a majority of S.

Proposition 12. *A process (that is not crashed) is disabled exactly when a majority of participants cease communicating with it,*

$$\text{DIS'D}(p) \;\Leftrightarrow\; \exists \mathsf{M} \subseteq \mathsf{S} \bigwedge_{q \in \mathsf{M}} \Box\neg\text{SEND}^*_q(p, +)\ .$$

Proof. For the "$\Rightarrow$" direction, it should be clear that safety is violated if fewer than a majority are needed. The "$\Leftarrow$" direction follows from Theorem 11. $\Box$

Proposition 13. *The fact "q is shunning p" is local to q, as is q's intent to eventually shun p,*

$$\begin{aligned}&a)\ K_r\text{SHUN}_q(p) \Leftrightarrow \text{RECV}^*_r(q, \mathit{ack}(\text{SHUN}_q(p)))\\ &b)\ K_r\Diamond\text{SHUN}_q(p) \Leftrightarrow \text{RECV}^*_r(q, \mathit{ack}(\diamond\text{SHUN}_q(p))).\end{aligned}$$

[7] In general, one requires a quorum; in a symmetric system that is a majority.

[8] Censoring q will also achieve the desired effect: q can only send messages to p that do not mention things that conflict with an attempt to disable p. For simplicity, we assume shunning involves all communication.

Proof. Similar to Proposition 2. □

Proposition 14. *A remote process r knows another process q has ceased communicating (resp. intends to cease communicating) with p if and only if q has explicitly mentioned* $\textsc{shun}_q(p)$ *(resp.* $\diamond\textsc{shun}_q(p)$*) to r,*

$$a)\ K_r \Box\neg\textsc{send}^*_q(p,+) \Leftrightarrow \textsc{recv}^*_r(q, ack(\textsc{shun}_q(p)))$$
$$b)\ K_r \Diamond\Box\neg\textsc{send}^*_q(p,+) \Leftrightarrow \textsc{recv}^*_r(q, ack(\diamond\textsc{shun}_q(p)))$$

Proof. a). The "$\Leftarrow$" direction is trivial. For the "$\Rightarrow$" direction, $K_r \Box\neg\textsc{send}^*_q(p,+)$ $\Leftrightarrow K_r(\textsc{shun}_q(p) \vee \textsc{crashed}(q))$. Now, assuming the contradiction, every $c' \sim_r c$ satisfies $\neg\textsc{recv}^*_r(q, \mathrm{ack}(\textsc{shun}_q(p))) \wedge (\textsc{shun}_q(p) \vee \textsc{crashed}(q))$. Construct c_1 from any such c' by rolling q back to the last event, call it e_q, that causally influenced r. Since r has not received a message originating at q indicating $\textsc{shun}_q(p)$, that event or q's crash, whichever holds on c, must have happened after e_q. However, $c_1 \sim_r c$ and $c_1 \models \neg\textsc{shun}_q(p) \ \wedge\ \neg\textsc{crashed}(q)$.

The proof of b) follows from Proposition 13 and the proof of a). □

Corollary 15. *Before a process r can know another process p will eventually be disabled, it must know a majority of p's cohorts will eventually shun p,*

$$K_r\Diamond\textsc{dis'd}(p) \Leftrightarrow K_r\Diamond\Big(\exists \mathsf{M} \subseteq \mathsf{S}\ \textstyle\bigwedge_{q\in\mathsf{M}} \Box\neg\textsc{send}^*_q(p,+)\Big) \Leftrightarrow$$
$$K_r\Big(\exists \mathsf{M} \subseteq \mathsf{S}\ \Diamond \textstyle\bigwedge_{q\in\mathsf{M}} \Box\neg\textsc{send}^*_q(p,+)\Big) \Leftrightarrow$$
$$\exists \mathsf{M} \subseteq \mathsf{S}\ \textstyle\bigwedge_{q\in\mathsf{M}} \textsc{recv}^*_r(q, ack(\diamond\textsc{shun}_q(p)))\ .$$

Proof. That r must know the majority exists (and not that it will eventually) follows by an argument similar to that of Proposition 6. □

Theorem 16. *Uniform Coordination is equally hard when channels guarantee delivery as when they do not.*

Proof. (Sketch. See [14]) Proposition 12 shows processes must invoke shunning to disable each other; disabling arises because of the desire to make exempt processes appear crashed, and this requires some mechanism by which processes can revoke another's ability to take actions. As a result, even when channels may guarantee delivery, no process can know a message it sends will be received – the destination may be shunning it – which is exactly the case when channels do not guarantee delivery. □

4.3 A Solution to Uniform Coordination

Figure 3 gives a solution to Uniform Coordination. The majority requirement for disabling a process percolates up to $\textsc{permit}_p(\alpha)$: a process that has not received ack('α')s from a majority cannot then disable that unresponsive majority. The message $\diamond$shun(T) is the request to shun processes in T, and the statement "await x messages" prevents a process from continuing until it has received at least x messages, though it may elect to wait for more responses.

/* Alg 4 : Initiator p */	/* Alg 4 : Respondent q - action */
choose α from task list; $Mcast_p(\mathsf{S}, \alpha)$; await $recv_p^*(q, `\alpha')$ for $q \in \mathsf{M} \subset \mathsf{S}$; $\mathsf{T} = \{q \in \mathsf{S} - \mathsf{M} \mid \neg K_p \Diamond \text{DIS'D}(q)\}$ $Mcast_p(\mathsf{M}, \diamond\text{shun}(\mathsf{T}))$ await $recv_p^*(q', \text{ack}(\diamond\text{shun}(\mathsf{T})))$ for $q' \in$ $\mathsf{M}' \subseteq \mathsf{M}$, still a majority of S /* $K_p \Diamond \text{DIS'D}(r)$ for $r \in \mathsf{S} - \mathsf{M}$ */ do α;	$recv_q^*(p, `\alpha')$; if promised $\diamond\text{SHUN}_q(p)$ then choose fairly   a. $send_q(p, \text{ack}(`\alpha'))$; add α to task list; b. begin shunning p; else $send_q(p, \text{ack}(`\alpha'))$; add α to task list; /* Alg 4 : Respondent q - shun */ $recv_q^*(p, \diamond\text{shun}(\mathsf{T}))$; if promised $\diamond\text{SHUN}_q(p)$ then choose fairly a. $send_q(p, \text{ack}(\diamond\text{shun}(\mathsf{T})))$; add T to shun list; b. begin shunning p; else $send_q(p, \text{ack}(\diamond\text{shun}(\mathsf{T})))$; add S to shun list;

Figure 3. Solution to Uniform Coordination.

A majority must agree to shun a process to actually disable it, but processes may begin shunning without further communication; while it must be done 'uniformly' a process does not need permission to take the action *shun*(p). This follows because, in contrast to $do_q(\alpha)$, q shunning r and q crashing both achieve the promised effect.

5 Conclusions

This paper describes necessary and sufficient conditions for a group of autonomous processes to take actions uniformly in asynchronous systems. In the absence of infinite storage space and failure detectors of guaranteed strength, communication and its lack are vital in ensuring both safety and liveness. That the relatively weak Uniformity problem requires majority agreement is due to the desire to *simulate* exempting processes from acting only if they have crashed; exempt processes must appear, for all intents and purposes, as if they had failed.

Terminal exemptions induce a primary partition behavior, but we can imagine coordination problems for which exemptions need not be terminal. These are exactly the problems in which progress can be made in multiple sides of a partition. Thus, it seems fruitful to understand coordination problems in terms of limiting the behavior of processes that, during certain periods, cannot be accounted for. Knowing what is possible during partitions is especially important in wide-area and cellular networks in which connectivity guarantees, due to cost and technology limitations, are less certain.

Finally, solutions to Uniform Coordination will eventually need to add processes and redefine obligation sets if runs are to be infinite. Since our exemption is terminal, once a minority are *either* crashed *or* disabled, we have reached the 'natural' limits. Individual problems will decide when and how to define the new obligation set to reflect the degree of system-wide coordination desired.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd FTCS*, pages 76–84. IEEE, 1992.
2. K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.
3. T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *11th PODC*, pages 147–158. ACM, 1992.
4. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS*, 3(1):63–75, 1985.
5. K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*, 1(1):40–52, 1986.
6. B. Chor and C. Dwork. Randomization in Byzantine Agreement. *Advances in Computer Research*, 5:443–497, 1989.
7. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374–382, 1985.
8. A. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In *3rd WDAG – LNCS 392*, pages 110–123. Springer Verlag, 1989.
9. J.Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *JACM*, 37(3):549–587, 1990.
10. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
11. L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
12. D. Malki, K. P. Birman, A. Ricciardi, and A. Schiper. Uniform Actions in Asynchronous Distributed Systems. In *13th PODC*, pages 274–284. ACM, 1994.
13. A. Ricciardi. Dissecting Distributed Coordination in Asynchronous Systems. Technical Report ECE-PDS-9502, The Univerisity of Texas, 1995.
14. A. Ricciardi. Sequential Distributed Coordination. Technical Report ECE-PDS-9503, The Univerisity of Texas, 1995.
15. A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *10th PODC*, pages 341–351. ACM, 1991.
16. L. Sabel and K. Marzullo. Simulating Fail-Stop in Asynchronous Distributed Systems. In *13th Symposium on Rel. Dist. Sys.*, pages 138–47. IEEE, 1994.
17. R. van Renesse, T. Hickey, and K. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1441, Cornell University, 1994.

This article was processed using the LaTeX macro package with LLNCS style