Knowledge-Theoretic Analysis of Partitionable Coordination

Aleta Ricciardi

TR TR-PDS-1997-013 December 1997

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

1 Introduction

Periodically coordinating specific actions among a group of processes is fundamental to most distributed computing problems, especially replication schemes that achieve fault tolerance. In asynchronous distributed systems coordination necessarily involves communication, and coordinating fault-tolerantly means doing so despite the presence of unresponsive members. The origins of unresponsiveness are not determinable in asynchronous systems, and therefore irrelevant; it is solely the inability to learn information about the unresponsive process that is problematic.

Fault-tolerant coordination is broadly and colloquially expressed as "all correct processes do x", which can be rephrased as "either a process does x, or it is faulty." One can consider the fact that a process is faulty as absolving it, or exempting it, from having to participate in the coordination. Obviously, unreachable processes (whether crashed or partitioned away) must be exempt from having to perform any activity. In large distributed systems, especially those prone to partitioning, it is unlikely that one can bound the number of unreachable processes. One contribution of this work is to show that in the absence of such a bound, this unimposing problem requires *perfect failure detectors* [5].

The *primary partition model* of execution permits progress within a quorum, or primary, subset of the coordinating processes; processes not in the primary partition, even though non-faulty, cannot make progress. Another contribution of this work is to define the exemption responsible for primary partition behavior. This exemption, which we call SIM-CRASH(p), gives rise to executions that are indistinguishable from those in which only faulty processes are exempt, yet it is knowable by processes in the system.

Another contribution of this work is that it suggests a simple, formal way to define coordination problems that can make safe progress in minority partitions. Having identified the primary partition exemption, numerous ways to weaken it are possible. While group-based middleware exists that permits progress in multiple minority partitions [1, 20], only recently [19] has there been a formal examination of problems that can do so safely. For distributed systems over networks prone to partitions, clearly defining *partitionable coordination problems* is crucial.

Finally, the central theorem of this paper highlights some significant differences between Uniform Distributed Coordination (UDC) [9] and non-uniform Distributed Coordination.

In Section 2 we describe the model of computation, the language, approach, and previous results we use to reason about it and to specify UDC. In Section 3 we define the problem and the notion of exemptions from coordination. Section 4 contains the knowledge analysis for UDC with the uninstantiated generic exemption, and further analyses for specific exemptions. Also in this section we define SIM-CRASH(), prove it is responsible for primary partition behavior, and suggest weaker exemptions for partitionable coordination.

2 Background

2.1 Model of Computation

We begin with the familiar model of an asynchronous distributed system: a finite set of processes, $\mathsf{Proc} = \{p_1, ..., p_n\}$ with no shared global clock. These processes communicate with one another by passing messages over a completely connected network of channels. Processes execute *events* in a totally ordered sequence. The communication events are $send_p(q, m)$ (p sends message m to q) and $recv_q(p, m)$ (q receives m from p). Processes fail by crashing and do not recover, but otherwise follow their assigned protocols. The event $crash_p$ models the failure of p, and can be followed only by other $crash_p$ events. The communication channels do not have a guaranteed bound on message transmission. They are reliable to the extent that messages between two processes that never crash are sure to be received; otherwise, any message only non-zero probability of reaching its destination. Channels do not garble or spontaneously generate messages.

A *history* for process p, denoted h_p, begins with the event $start_p$, which occurs exactly once. A *run over* Proc is a tuple of infinite process histories, one for each $p \in \mathsf{Proc}$; the history of a crashed process has a trailing sequence of infinitely many *crash* events. A run is well-formed if it reflects the assumptions of our model; *e.g.*, it contains the corresponding send event for every receive event it contains, $crash_p$ events are followed only by other $crash_p$ events, and so forth. We only consider well-formed asynchronous runs over Proc.

A process is correct in a run if does not crash; otherwise it is faulty. A *cut over* Proc is a tuple of finite process histories, one for each $p \in \mathsf{Proc}$. A cut $c = (h_1, \ldots, h_p, \ldots, h_n)$ is consistent if and only if, for every $recv_q(p, m)$ event it contains, it also contains the corresponding $send_p(q, m)$ event [10]. A run r *completes* a cut c if and only if c is a prefix of r (written $c \in r$).

Consistent cuts in a run are naturally partially ordered by comparing prefixes of the component histories. Given $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$, both in the same run r, $c \leq c'$ (and $c < c'$) if and only if for each $p \in \mathsf{Proc}$, h_p is a prefix of h'_p (and $c \neq c'$). Consistent cuts, not necessarily in the same run, can also divided into equivalence classes according to process histories. Given $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$, $c \sim_p c'$ if and only if $h_p = h'_p$.

2.2 Formal Language

Our formal language for expressing distributed coordination is linear time temporal logic [2], while the language used for analyzing solutions also involves knowledge modalities [7]. All formulas are evaluated on consistent cuts, in the context of a run. Such a pair (r, c) is called a *point*. Since $\sim_p$ is insensitive to other processes' histories, it is trivial to extend the relation to points, $(r, c) \sim_p (r', c')$.

2.2.1 Semantics of Modal Operators

For φ a formula of our language, we write $(r, c) \models \varphi$ to denote that φ is true at the point (r, c). Base propositional formulas include $\text{SEND}_p(q, m)$, which is true at point (r, c) precisely when $send_p(q, m)$ is an event in p's history component of c. Analogous statements hold for the formulas $\text{RECV}_q(p, m)$ and $\text{CRASH}(p)$.

The semantics of *temporal* modal operators are: $(r, c) \models \Box\varphi$ if and only if for each $c' \in r$, such that $c \leq c'$, we have $(r, c') \models \varphi$. The standard interpretation is that φ holds at (r, c) and throughout the rest of the run. The dual of $\Box\varphi$ is $\Diamond\varphi$ which is interpreted as φ holding at some, but not every, future point (r, c') The *knowledge* modal operator is $K_p\varphi$ (p knows φ). The semantics capture the idea that, given its current history, p cannot imagine φ being false: $(r, c) \models K_p\varphi$ if and only if for all points $(r', c') \sim_p (r, c)$, we have $(r', c') \models \varphi$. Note that crashed processes can indeed know facts; however, since they cannot issue further send events, they cannot communicate their knowledge should the need arise.

2.2.2 Knowledge Acquisition

The analyses in Section 4 use results from Chandy and Misra describing how one process learns facts about other processes [6]. A formula is said to be *local* to process p if at every point p knows whether it is true. Thus, all formulas describing a process's local state, for example $\text{SEND}_p(q, m)$, $\text{RECV}_q(p, m)$ and $\text{CRASH}(p)$, are local to only that process. The axioms for the modal logic S5 also mean that knowledge is local to the knower: for each formula $\varphi : K_p(K_p\varphi) \vee K_p\neg(K_p\varphi)$.

Chandy and Misra show that for formulas local to only one process, others' knowledge of them is attained only by communicating with that process. Because $\text{CRASH}(p)$ is local only to p, Proposition 1 underscores the impossibility of any other process ever knowing that p is crashed. We rephrase the crucial knowledge-acquisition theorem for our purposes.

Proposition 1 *[Chandy-Misra] Suppose φ is local only to p. Let (r, c) and (r, c') be two points with $c < c'$. If $(r, c) \models \neg K_q\varphi$ and $(r, c') \models K_q\varphi$, then c' contains a chain of messages* *that originated with p and terminates with q, and that implies the continued truth of φ.*

Proposition 2 *Suppose φ is local to $s \neq q$.*[1] *Then in the absence of a known bound on the number of process failures, $K_p \Diamond K_q \varphi \Rightarrow K_p K_q \varphi$.*

PROOF By Proposition 1, if $K_q\varphi$ ever holds, q must have received a message from s stating φ. Overusing notation to allow the formula to also represent a message whose contents implies that formula, if $(r, c) \models K_p \Diamond K_q \varphi$ then $\forall (r', c') \sim_p (r, c) : (r', c') \models \Diamond \text{RECV}_q(s, \varphi)$. For every such p-indistinguishable point (r', c') there are three cases:

1. $(r', c') \models \text{RECV}_q(s, \varphi)$. Process q has already received the message.
2. $(r', c') \models \text{SEND}_s(q, \varphi) \wedge \neg\text{RECV}_q(s, \varphi)$. The required message from s is in transit.

In this case, construct $c'' \sim_p c$ which is identical to c' except that $h''_q = h'_q \cdot \mathit{crash}_q$. Such a cut cannot satisfy $\Diamond\text{RECV}_q(s, \varphi)$. That is, not every pair satisfying $\text{SEND}_s(q, \varphi) \wedge \neg\text{RECV}_q(s, \varphi)$ also satisfies $\Diamond\text{RECV}_q(s, m)$.

3. $(r', c') \models \neg\text{SEND}_s(q, \varphi)$. The required message from s has not yet been sent.

Again, we can construct $c'' \sim_p c$ on which $\Diamond\text{RECV}_q(s, \varphi)$ does not hold, either because $h''_s = h'_s \cdot \mathit{crash}_s$ or because $h''_q = h'_q \cdot \mathit{crash}_q$. For the special case when $s = p$, the cut c'' with $h''_p = h_p \cdot \mathit{crash}_p$ is not p-indistinguishable from h_p. However, for some r'', $(r'', c'') \models \Box\neg\text{SEND}_p(q, \varphi)$, and the impossibility of q ever receiving φ follows from there. As in case 2, not every pair satisfying $\neg\text{SEND}_s(q, \varphi)$ also satisfies $\Diamond\text{RECV}_q(s, m)$. ■(step 3)

Since the second and third cases both admit p-indistinguishable subsets of points that do not satisfy $\Diamond\text{RECV}_q(p, \varphi)$, it must be that whenever $K_p \Diamond K_q \varphi$ neither of these cases are possible. That is, whenever $K_p \Diamond K_q \varphi$ holds, every p-indistinguishable point satisfies $\text{RECV}_q(p, \varphi)$, so $K_p K_q \varphi$. $\boxed{\text{QED}}$

Another way to understand Proposition 2 is that p's knowledge of q's eventual knowledge of a fact implies p's knowledge that neither will crash, or that no more than f processes will crash.

Proposition 3 *Suppose φ is not local to q. If $(r, c) \models \neg K_q \varphi$ then there are runs completing c throughout which q is correct and never learns φ:*

$$\Big\{(r', c) \mid (r', c) \models \Box\neg K_q \varphi \wedge \Box\neg\text{CRASH}(q)\Big\} \neq \emptyset \;.$$

PROOF Given $c = (h_1, \ldots, h_p, \ldots, h_n)$, as described, consider the subset of completions in which q is correct. Construct c' from c by immediately crashing every process that knows

[1] This can be extended to $S \subset \mathsf{Proc}$. If it is assumed that at most f processes can crash and S has more than f processes, then the proposition does not hold.

φ. If c were such that no messages from any of the newly-crashed processes were in transit to q, then q can never learn φ. If there were messages in transit to q, then there are some completions of c' in which each such message is lost since the channel guarantee holds only for correct processes. $\boxed{\text{QED}}$

3 Uniform Distributed Coordination

By *distributed coordination* among Proc, we mean that the members of Proc are attempting to each execute certain *actions* of interest. The actions may be allocating a resource, delivering multicast messages, or committing a transaction, though we are not concerned with the specifics. We are also not concerned here with other requirements such as executing actions in a particular order (*e.g.*, total-order multicast), or not executing conflicting actions (*e.g.*, consensus). We are only interested in the eventual, distributed execution of these actions. The event whereby p executes action α is denoted $do_p(\alpha)$. As with other events processes execute, the propositional formula $\text{DO}_p(\alpha)$ holds at point (r, c) exactly when $do_p(\alpha)$ is an event of h_p, and $\text{DO}_p(\alpha)$ is local only to p.

For distributed systems that can experience only benign failures, the *uniform* statement of coordination [9] is commonly used. Colloquially, Uniform Distributed Coordination of actions, $\alpha \in \mathcal{A}$, among processes $p \in \mathsf{Proc}$ specifies that if any $p \in \mathsf{Proc}$ executes $\alpha \in \mathcal{A}$ then every correct $q \in \mathsf{Proc}$ eventually also executes α. This is written as,[2]

$$p \in \mathsf{Proc} \wedge \text{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Diamond\Big(\text{DO}_q(\alpha) \vee \text{CRASH}(q)\Big) \ . \tag{1}$$

Each action $\alpha \in \mathcal{A}$ is associated with a single process, p_α, that initiates its distributed execution. The event whereby p_α first becomes aware that α should be executed is denoted $try_{p_\alpha}(\alpha)$. The formula $\text{TRY}(\alpha)$ holds at (r, c) exactly when $try_{p_\alpha}(\alpha)$ is an event in h_{p_α}. Since p_α is unique, we have $\text{TRY}(\alpha) \Leftrightarrow K_{p_\alpha}\text{TRY}(\alpha)$ and $K_p\text{TRY}(\alpha) \Rightarrow (p = p_\alpha) \vee \text{RECV}_p(p_\alpha, \text{TRY}(\alpha))$.[3]

Validity in UDC requires that only actions arising in a run be taken, not every action enumerable in $\mathcal{A}$, $\text{DO}_p(\alpha) \Rightarrow K_p\text{TRY}(\alpha)$. Liveness requires correct processes to eventually execute all actions of which they become aware, $K_p\text{TRY}(\alpha) \Rightarrow \Diamond\Big(\text{DO}_p(\alpha) \vee \text{CRASH}(\alpha)\Big)$.

This simple problem is equivalent to *Uniform Reliable Multicast* [18] by identifying $do_p(\alpha)$ with the reliable delivery event. We show in Section 4.1 that this problem requires processes

[2] Non-uniform Distributed Coordination has the consequent $\forall q \in \mathsf{Proc} : \Diamond(\text{DO}_q(\alpha) \vee \text{CRASH}(q) \vee \text{CRASH}(p))$.

[3] Technically, $\text{TRY}(\alpha)$ is not local to p_α since in each run any process may be the unique initiator of any action. As such, when $\text{TRY}(\alpha)$ does not hold, no p actually knows this, $\neg\text{TRY}(\alpha) \Rightarrow \forall p \in \mathsf{Proc} : \neg K_p \neg\text{TRY}(\alpha)$. While Chandy and Misra explicitly avoid such formulas, with simple restrictions their results apply.

to have *perfect failure detectors* [5] when the number of faulty processes in not bounded in advance. This fits precisely with the presentation in [18] in which the Isis virtual synchrony model [3] is assumed; in Section 4.2 we show that the underpinnings of this model are *simulated* perfect failure detectors [17].

Exemptions from Coordination

To define a richer set of problems, that may possibly have solutions in this and other environments, we generalize *exemptions from coordination*. In Formula 1, a process is not required to execute α only if it is crashed. We generalize UDC as follows:

$$p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Diamond\Big(\textsc{do}_q(\alpha) \vee \textsc{exempt}_q(\alpha)\Big) \quad (2)$$

The semantics of $\textsc{exempt}_q(\alpha)$ and the ramifications for UDC depend on how it is instantiated. For example, when $\textsc{exempt}_q(\alpha)$ is the simple truth formula $\top$, no real coordination is achieved because the consequent in Formula 2 is trivially satisfied. A variant of UDC is *non-trivial* if $\textsc{exempt}_q(\alpha)$ local to at most q; it may be local to no process. To maintain the distinction with non-uniform Distributed Coordination, we require that for no p does $\textsc{crash}(p)$ imply $\textsc{exempt}_q(\alpha)$.[4] At the other side of the spectrum, instantiating $\textsc{exempt}_q(\alpha)$ with $\perp$ precludes solutions if even one process can crash. Instantiating $\textsc{exempt}_q(\alpha)$ with $\textsc{crash}(q)$ gives Formula 1. Because Liveness is specified in terms of processes crashing, it is also affected by the particular instantiation of $\textsc{exempt}_q(\alpha)$.

4 Knowledge Analysis

Using the modal logic of knowledge to analyze problems and algorithms in distributed computing is efficient, if rather specialized [6, 7, 12, 14, 16]. Knowledge analyses are based on the fact that an algorithm is a solution to a problem if and only if the formulas specifying the problem are *valid* over the set of all runs generated by the algorithm. For valid formulas, one can apply the modal logic *generalization inference rule* which introduces the modal operator, in this case K_p. Starting with UDC with abstract exemptions, knowledge analyses proceed as follows:

1. Apply modal logic generalization to Formula 2,

$$K_p\Big(p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha) \Rightarrow \big(\forall q \in \mathsf{Proc} : \Diamond\textsc{do}_q(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\big)\Big)$$

[4] In non-uniform Distributed Coordination (note 2) we would have $\textsc{exempt}_q(\alpha) \overset{\text{def}}{=} \textsc{crash}(q) \vee \textsc{crash}(p)$.

2. Apply axioms and tautological reasoning of S5 (K_p distributes over both $\Rightarrow$ and $\wedge$),

$$\Big(K_p\big(p \in \mathsf{Proc}\big) \wedge K_p\textsc{do}_p(\alpha)\Big) \Rightarrow \forall q \in \mathsf{Proc} : \Big(K_p\big(\Diamond\textsc{do}_q(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\big)\Big)$$

3. Local formulas in step 2 antecedent,[5] $K_p(p \in \mathsf{Proc}) \wedge K_p\textsc{do}_p(\alpha) \Leftrightarrow p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha)$.

4. Apply temporal logic tautology ($\Diamond$ distributes over $\Rightarrow$) to UDC Validity, $\Diamond\textsc{do}_q(\alpha) \Rightarrow \Diamond K_q\textsc{try}(\alpha)$

5. We have the intermediate statement

$$p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : K_p\Big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\Big) \ .$$

6. Accountability. We prove this statement in Proposition 4, though state it here for continuity of the knowledge analysis. The knowledge operator does not normally distribute over disjunction, but in this particular case, for this problem it does.

$$K_p\Big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\Big) \Leftrightarrow K_p\Diamond K_q\textsc{try}(\alpha) \vee K_p\Diamond\textsc{exempt}_q(\alpha) \ .$$

7. Apply Proposition 2, $K_p\Diamond K_q\textsc{try}(\alpha) \vee K_p\Diamond\textsc{exempt}_q(\alpha) \Leftrightarrow K_pK_q\textsc{try}(\alpha) \vee K_p\Diamond\textsc{exempt}_q(\alpha)$.

8. Knowledge of actions is local to only the knower, because $K_q\textsc{try}(\alpha) \Rightarrow \textsc{recv}_q(p_\alpha, \textsc{try}(\alpha))$ and because $\textsc{recv}_q(p_\alpha, \textsc{try}(\alpha))$ is local only to q. $K_pK_q\textsc{try}(\alpha) \Leftrightarrow \textsc{recv}_p(q, K_q\textsc{try}(\alpha))$.

9. Finally, we have, $p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Big(\textsc{recv}_p(q, K_q\textsc{try}(\alpha)) \vee K_p\Diamond\textsc{exempt}_q(\alpha)\Big)$.

Proposition 4 *[Accountability] Knowing that another process will eventually learn about an action or eventually be exempt from executing it, necessarily means definitive knowledge of one or the other.*

$$K_p\Big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\Big) \Leftrightarrow K_p\Diamond K_q\textsc{try}(\alpha) \vee K_p\Diamond\textsc{exempt}_q(\alpha)$$

[5] Since Proc is static, locality of membership follows for each p. This makes obvious why dynamic coordination problems, which continually coordinate despite changes in the set of processes coordinating, are more complex.

PROOF The $\Leftarrow$ direction is trivial. We prove the $\Rightarrow$ direction by contradiction. Consider a point (r,c) such that $(r,c) \models K_p\big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\big) \wedge \neg K_p \Diamond K_q\textsc{try}(\alpha) \wedge \neg K_p\Diamond\textsc{exempt}_q(\alpha)$. Then for any $(r',c') \sim_p (r,c)$ four cases are combinatorially possible; we show the last two are logically impossible.

1. $(r',c') \models \Diamond K_q\textsc{try}(\alpha) \wedge \Diamond\textsc{exempt}_q(\alpha)$
2. $(r',c') \models \Box\neg K_q\textsc{try}(\alpha) \wedge \Diamond\textsc{exempt}_q(\alpha)$
3. $(r',c') \models \Box\neg K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha)$

   This is not possible since $K_p\big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\big)$.
4. $(r',c') \models \Diamond K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha)$

We show that for each such p-indistinguishable point, there are other p-indistinguishable points that satisfy $\Box\neg K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha)$. So consider $(r',c') \sim_p (r,c)$, with $c' = (h'_1,\ldots,h'_p,\ldots,h'_n)$, and $(r',c') \models \Diamond K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha) \wedge \neg K_p\Diamond K_q\textsc{try}(\alpha)$. Since $q \neq p_\alpha$ (otherwise $K_pK_q\textsc{try}(\alpha)$ would hold), Proposition 2 gives $(r',c') \models \neg K_pK_q\textsc{try}(\alpha)$. Given that, Proposition 1 means that p has received no message from q indicating $K_q\textsc{try}(\alpha)$. Therefore, h'_q is of the form, $h'_q = \mathit{start}_q \cdots \mathit{send}_q(s,m) \cdots e_q$, where $\mathit{send}_q(s,m)$ is the last event in h'_q that causally affects p in $h'_p = h_p$. If $K_q\textsc{try}(\alpha)$ holds at (r',c'), q learned $\textsc{try}(\alpha)$ after $\mathit{send}_q(s,m)$.

From c', define c'' which differs from c' in that q's history is truncated after the event $\mathit{send}_q(s,m)$. If c'' is not consistent, it can be made so by truncating other processes' histories but still without affecting h_p, since no event after $\mathit{send}_q(p,m)$ causally affects any event in h_p. Since histories have only been truncated, $c'' \in r'$ and $(r',c'') \models \Diamond\Box\neg\textsc{exempt}_q(\alpha)$.

By construction, c'' does not contain the event $\mathit{recv}_q(p_\alpha, \textsc{try}(\alpha))$, so $(r',c'') \models \neg K_q\textsc{try}(\alpha)$. Also by construction, $(r',c'') \sim_p (r,c)$. From Proposition 3, there are completions of c'' throughout which q never crashes, but never learns $\textsc{try}(\alpha)$ solely because of the failure of all processes that know it at point (r',c''). Let $S = \{s \in \mathsf{Proc} \mid (r',c'') \models K_s\textsc{try}(\alpha)\}$.

To complete this step, we must show that $\Diamond\textsc{exempt}_q(\alpha)$ does not necessarily hold in these completions of c''. This follows from the restrictions on $\textsc{exempt}_q(\alpha)$, which cannot be local to processes other than q, and is not entailed only by $\textsc{crash}(S)$.

Thus, for any p-indistinguishable point (r',c') that might satisfy $\Diamond K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha)$, there are p-indistinguishable points (r'',c'') that satisfy $\Box\neg K_q\textsc{try}(\alpha) \wedge \Box\neg\textsc{exempt}_q(\alpha)$. ∎(step 4)

Since the third case is impossible and fourth case admits p-indistinguishable subsets of points that do not satisfy $\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)$, it must be that whenever $K_p(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha))$ holds, neither of these cases are possible. Every p-indistinguishable point therefore satisfies $\Diamond\textsc{exempt}_q(\alpha)$, and we conclude $K_p\big(\Diamond K_q\textsc{try}(\alpha) \vee \Diamond\textsc{exempt}_q(\alpha)\big) \Rightarrow K_p\Diamond\textsc{exempt}_q(\alpha)$.

Or-introduction is now possible, the most meaningful being $K_p \Diamond K_q \textsc{try}(\alpha)$ as it retains the tautological direction of the proof,

$$K_p\Big(\Diamond K_q \textsc{try}(\alpha) \vee \Diamond \textsc{exempt}_q(\alpha)\Big) \Rightarrow K_p \Diamond \textsc{exempt}_q(\alpha) \vee K_p \Diamond K_q \textsc{try}(\alpha)\ . \quad \boxed{\text{QED}}$$

Proposition 5 *The communication requirements for safely taking an action α, as specified by UDC are,*

$$p \in \mathsf{Proc} \wedge \textsc{do}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Big(\textsc{recv}_p(q, K_q \textsc{try}(\alpha)) \vee K_p \Diamond \textsc{exempt}_q(\alpha)\Big)\ .$$

Proof This follows by applying Proposition 2 and then Proposition 1 to Proposition 4. $\boxed{\text{QED}}$

4.1 Consequences of Proposition 5

The proof of Proposition 4 sheds light on the difference between non-uniform Distributed Coordination and UDC. First, notice that the fourth case in the proof becomes tenable if the failure of processes other than q (*i.e.*, those knowing $\textsc{try}(\alpha)$) causes q to be exempt. This is precisely the case of non-uniform Distributed Coordination as described in note 2. This means that a process may execute actions without first communicating with others, comfortable in the knowledge that should it fail, the others are not obliged to execute an action they cannot possibly learn about.

It is argued [9] that UDC is better suited to benign failures, and non-uniform Distributed Coordination to malicious failures; in the latter case, it is not desirable to have otherwise 'loyal' processes follow the actions of a 'traitor', nor is it possible to assure that a malicious process will follow its assigned protocol. Neither argument applies to benign failures. The non-uniform variant is exactly the specification of the Byzantine Generals Problems [11] where 'loyal' processes must all decided the same value, which must be the general's value provided the general is not a 'traitor'. That is $\textsc{do}_g(decide_g(v_g)) \wedge \textsc{byz-faulty}(g) \Rightarrow \textsc{exempt}_q(decide_q(v_g))$; the faultiness of the acting process, the general g, necessarily causes other processes to be exempt.

A second important aspect of Proposition 4 is that for UDC, it holds irrespective of any non-trivial exemption. Recall that an exemption is non-trivial if it is local to at most the exempt process. It states explicitly that communication between coordinating processes, or in its absence knowledge of eventual exemption are required before any action can be executed. The difficulty of the problem therefore depends solely on the particular instantiation of $\textsc{exempt}_q(\alpha)$ as this dictates the communication required to learn it.

Thirdly, for problems traditionally stated in terms of the *correct processes* coordinating uniformly, when $\text{EXEMPT}_q(\alpha)$ is instantiated with $\text{CRASH}(q)$, Proposition 4 means that processes must have *perfect failure detectors* [5].

Corollary 1 *Any solution to Uniform Distributed Coordination as specified by Formula 1 in which the number of failures that may occur is not bounded a priori requires perfect failure detectors.*

No weaker failure detector, not even an eventually perfect failure detector, suffices to permit $K_p\text{CRASH}(q)$ to hold; since predicting failures is ruled out, $K_p\Diamond\text{CRASH}(q)$ cannot hold either.

A fourth consequence, related to the third, is the significance of having no *a priori* bound on the number of failures Proc may experience. With an upper limit of say f, then Proposition 3 can be circumvented. It is possible for $K_p(\Diamond K_q\text{TRY}(\alpha) \vee \Diamond\text{EXEMPT}_q(\alpha))$ to hold, without p knowing either one explicitly, provided p knows more than f processes know $\text{TRY}(\alpha)$ and provided that crashed processes are exempt. If q never crashes, at least some process knowing $\text{TRY}(\alpha)$ will also never crash and its knowledge will eventually reach q; if q does crash it will be exempt. Not bounding the number of failures is equivalent to $f = n-1$, and so dissemination of $\text{TRY}(\alpha)$ must be to every member of Proc.

4.2 Simulated Crash Failure and Simulated Failure Detectors

A fifth important consequence of Proposition 5 and Corollary 1 is that they provide a straightforward way to tweak UDC. By defining exemptions weaker than $\text{CRASH}(q)$, that are both knowable and non-trivial, one can thereby define useful variants of coordination that are solvable. In this section, we define the exemption $\text{SIM-CRASH}(p)$, and prove it is equivalent to primary partition behavior for UDC. For partitionable coordination in which progress is permitted in minority subsets, the search would be for still weaker exemptions.

The proven impossibility of Consensus [8] does not lessen the need to solve the problem. In fact, within a few years of the theoretical results, distributed middleware was being built that seemed to continually 'solve' Consensus [1, 4, 13, 15]. Chandra's and Toueg's failure detectors [5] significantly clarified our understanding of the choices implementers have, and the ramifications of their choices. Marzullo and Sabel then defined *simulated* perfect failure detectors [17] which present applications with black boxes that are indistinguishable from perfect failure detectors. They provide simulated Strong Accuracy and restricted, because the simulated failure detector itself may not be live, Strong Completeness. Essentially, simulated failure detectors make reality conform to their suspicions before notifying the application, in contrast to 'classic' failure detectors which are constrained to report on reality.

In retrospect, building and using simulated perfect failure detectors is precisely what is done by these middleware offerings to implement *virtual synchrony* [3] or its variants.

In the terms we have introduced here, a processes that is exempt from coordinating is made to appear, for all intents and purposes, as if it had crashed. For distributed coordination, the relevant factor is whether the exempt process can continue to execute actions from the set $\mathcal{A}$. We define $\textsc{sim-crash}(p)$ as $\forall \alpha \in \mathcal{A} : \big(\neg\textsc{do}_p(\alpha) \Rightarrow \Box\neg\textsc{do}_p(\alpha)\big)$. As $\textsc{crash}(p) \Rightarrow \textsc{sim-crash}(p)$, but not vice versa, it may be possible for processes to know this weaker exemption. In fact, a simulated perfect failure detector waits until it knows $\textsc{sim-crash}(p)$, then issues its 'detection'.

4.2.1 Making Progress and the Necessity of Shunning

With respect to UDC, ensuring $\Diamond\textsc{sim-crash}(q)$ means that the q's ability to continue taking actions from $\mathcal{A}$ must be revoked. Examining Proposition 4, it will be impossible for q to execute α if there is any $s \in \mathsf{Proc}$ from which q never receives the required message stating $K_s\textsc{try}(\alpha)$ and for which q can never learn $\Diamond\textsc{sim-crash}(s)$. Since communication between members of Proc is evidently necessary in solving UDC, it should not be surprising that the absence of communication can be used to prevent processes from executing actions. For s to ensure q never receives further messages from it, s should stop sending messages to q. We say s is *shunning* q. We denote the event whereby s begins shunning q by $shun_s(q)$, and note that as usual $\textsc{shun}_s(q)$ is local only to s.

4.2.2 The Primary Partition Exemption

Having some process s shun q is not, in itself, sufficient to ensure $\textsc{sim-crash}(q)$; q must also be prevented from learning $\Diamond\textsc{sim-crash}(s)$. The following proposition follows easily from the definitions of quorum and shunning.

Proposition 6 *A quorum of processes shunning q is necessary and sufficient to ensure $\textsc{sim-crash}(q)$ whenever q is not crashed.*

Informally, primary partition behavior refers to the restriction of infinite execution histories to a unique subset of processes; processes not in this subset, even though correct, will execute only finitely many events. We conclude that $\textsc{sim-crash}(q)$ gives rise to primary partition behavior in UDC and that this is necessarily achieved by shunning.

Combining Proposition 6 with Proposition 4, we see that in attempting to execute α, each process p must first receive messages acknowledging $\textsc{try}(\alpha)$ from at least a quorum of Proc; p can have no hope of learning $\Diamond\textsc{sim-crash}()$ for each member of a quorum. Having passed the first hurdle, p can try to learn $\diamond$SIM-CRASH(q) for any q from which it has not yet received a message.

Provided a quorum of processes are able to and continue to communicate with one another, they may still take actions. Of course, a minority of sim-crashed processes together with a single real crash will prevent further progress. But in the absence of perfect failure detectors, UDC with the SIM-CRASH() exemption provides runs indistinguishable from those in which an actual perfect failure detector is available.

4.2.3 Exemptions for Partitionable Coordination

It is often reasonable to allow processes in a minority partition to continue taking actions. An simple example occurs when the minority partition itself represents a quorum of processes with respect to some specific shared variable, as commonly occurs in replicated databases. For example, suppose logical variables x and y are replicated at processes $\{p_1, p_2, p_3\}$ and $\{p_3, p_4, p_5\}$ respectively, and a temporary network partition occurs to sever $\{p_1, p_2, p_4\}$ from $\{p_3, p_5\}$. Then the former group should still be able to update x, and the latter should still be able to update y. This example is not wholly satisfying as it can be thought of as the union of many primary partition problems.

SIM-CRASH(q) requires that q never take another action once it is exempt. However, one can imagine an exemption that does not necessitate the cessation of all future activity, but only of some activity, presumably related to the action that was not taken. The current work by Sussman and Marzullo on partitionable coordination takes this approach; in their work only a restricted set of actions are permitted in minority partitions [19].

Exemptions could also be defined that preclude activity, but only until some other condition is met. In the replicated database example, the condition would be obtaining the current value of the logical variable; when the network partitions are resolved, p_4's copy of y will have to be updated, as will p_3's copy of x.

5 Conclusions

We presented a knowledge-theoretic analysis of Uniform Distributed Coordination that highlights the differences between uniform and non-uniform variants of the problem, and the difficulty in having no *a priori* bound on the number of process failures the coordinating processes may suffer. In addition to the consequences already, these analyses explicitly describe the *minimal communication requirements* of any solution to the problem, or the exact reason for its impossibility.

The generalized form of UDC has allowed us to begin unifying a growing body of work in classic failure detectors, simulated failure detectors, various impossibility results based on these, and numerous implementations of virtual (or view) synchrony. By providing a generalized formal description of the basis for coordination, we expect that safe partitionable coordination can be better understood.

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] H. I. Banieqbal, H. I. Barringer, and A. Pneuli (editors). *Temporal Logic in Specification.* Springer Verlag, 1987. LNCS 398.

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

[4] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

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

[7] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge.* The MIT Press, 1995.

[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. Springer Verlag (LNCS 392)*, pages 110–123, 1989.

[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] M. Mazer. *A Knowledge-Theoretic Account of Negotiated Commitment.* PhD thesis, University of Toronto, 1989.

[13] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-tolerant Distributed Programs. *Distributed Systems Engineering Journal*, 1(2):87–103, 1993.

[14] G. Neiger and R. Bazzi. Using Knowledge to Optimally Achieve Coordination in Distributed Systems. In *Fourth TARK*, pages 43–59, 1992.

[15] L. L. Peterson, N. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM TOCS*, 7(3):217–246, 1989.

[16] A. M. Ricciardi. Practical Utility of Knowledge-Based Analyses. In *Fourth TARK*, pages 15–28, March 1992.

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

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

[19] J. Sussman and K. Marzullo. The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asychronous System. Technical report in preparation, 1998.

[20] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, 1996.