Sequential Distributed Coordination

Aleta Ricciardi

TR TR-PDS-1995-003 May 1995

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

Abstract

We give a broad characterization of distributed coordination, building on a basic agreement clause, then defining a generic sequence clause and a generic termination clause. We instantiate these to obtain common problems. Our decomposition isolates communication costs due solely to ordering or termination, giving a clear way of comparing coordination problems. We further explore the notion of exempting processes from participating in the coordination, and show that any problem in which exempt processes must be made to appear as if they had crashed is equally hard irrespective of channel properties (lossy or reliable). While somewhat surprising, this result also exposes why coordination is difficult, and reasonable ways to weaken it for systems prone partitioning. **Keywords:** Asynchronous, coordination, failures, sequence, termination.

1 Introduction

This paper builds on our earlier work analyzing Distributed Coordination in asynchronous systems [17, 21]. Uniform Coordination is broadly stated as requiring all processes (that never stop executing) to take a specific action if any process takes that action. Here we examine the communication required to solve problems in which the coordinated actions must also be taken in some consistent order, and those in which actions must be known to have been taken at all required sites. We define a generic *Sequence* condition and instantiate it with a wide range of formulas, giving familiar and commonly used orderings. We also define a generic *Termination* condition, and instantiate it to get common notions of *virtual synchrony* [4]. In addition to these familiar problems, separating sequence and termination requirements from basic agreement facilitates defining new coordination problems that may be appropriate for (*i.e.*, have solutions in) non-traditional distributed computing domains.

Our approach is similar to that of Hiltunen and Schlichting [15] who analyze properties of membership services. They give a number of conditions defining the sequence in which membership change messages may arrive at an application, as well as the sequence in which these messages are interleaved with other application messages. The latter defines several notions of virtual synchrony, and can be understood as timeliness guarantees. Our paper considers generic distributed coordination problems, defining variations of sequence and timeliness properties. As such, it can be viewed as generalizing theirs approach. As in [15], our hope is that understanding generic coordination will quantifiably differentiate existing problems and will give rise to new coordination problems. With respect to the latter, our concern is with networks (*e.g.*, cellular and wide-area) prone to partition, as previous results have shown many coordination problems inappropriate in these domains. As well, we expect the requisite properties of a membership service for a given instance of distributed coordination will then be evident. That is, it seems likely there is a natural mapping of coordination properties to membership service guarantees (first mentioned in [24]). In this way the underlying services defines a new metric for assessing the difficulty of distributed problems.

This task-oriented, modular specification allows us to factor out conditions unrelated to ordering actions consistently. For example, adding the sequence requirement on top of a basic, well-understood coordination requirement shows the exact cost of sequencing actions, over and above simple distributed coordination. The strengths of this modular approach lie in gleaning performance for distributed applications, especially important as they are run over wide-area or other costly networks, and in quantifying the information that must pass from one participant to another. Our approach, therefore, facilitates correct, efficient, specification-driven solutions.

We adopt the view that distributed coordination generally involves a set of designated *actions* to be taken, a way of defining which processes are *obliged* to participate in the coordination, and conditions under which some of them may be *exempt* from taking some of the actions. For example, any coordination problem must exempt crashed processes from taking actions, and this, traditionally has been the only exemption considered. This strict exemption requires processes to accurately identify crashed cohorts and, in asynchronous systems, this is not possible [13]. The coordination achieved in Isis [7], Horus [25], and Transis [1] finesses this by weakening the exemption so that agreement problems need not identify crashed processes accurately; instead, these systems create runs indistinguishable from ones in which an exempt process did in fact crash. It is beyond the scope of this paper to examine weaker exemptions (see [22]); here, we consider only exemptions that mimic process failure and give rise to *primary partition* behavior.

Section 2 describes our model of computation, while Section 3 defines Uniform Coordination and briefly presents the relevant results of our earlier work. We further examine process exemptions and prove that any problem in which exempt processes must appear as if they were crashed is no easier when channels are reliable than when they may lose messages. In Sections 3.2 and 3.3, we define the generic Sequence and Termination conditions, and formalize common examples from the literature. Sections 4 and 5 contain results on the minimal amount of communication needed to attain different ordering and termination conditions. We conclude in Section 6

2 Asynchronous System Model

The broader system consists of a infinite domain of process identifiers, Proc, though at any point in time only a finite subset of them are executing. The name space is infinite to model infinite system executions in which processes continually fail, terminate, are born, and recover. Processes communicate with each other only by passing messages through a network of channels. 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, which we model by the distinct event $crash_p$, but otherwise follows its assigned protocol.

Channels may lose messages (though not corrupt them), but every message sent has some non-zero probability of being received. To compensate for this, a process will resend a message until it is acknowledged. Since each message has some probability of being delivered at its destination, if two processes p and q both never fail, every message from p will eventually reach q (and vice versa);

however, if p crashes, messages may be lost. In this environment no process can determine whether an unresponsive process has crashed, or whether transient performance factors make it appear to have crashed [13].

Each process has local state and a transition function; a change in a process's local state is marked by an *event* in the execution of the process. We assume that after each event in its history, 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 unique 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 familiarity with inter-event causality (Lamport's $\rightarrow$ relation [16]) and consistent cuts [9]. A *system run* is a tuple of infinite process histories, one for each process that ever executes.[1] A cut, then, is a finite prefix of a run. 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, $\mathsf{S}_0 \stackrel{\text{def}}{=} \left\{p \mid \nexists q : start_q \rightarrow start_p\right\}$. The initial cut consists only of all the $start_p$ events for $p \in \mathsf{S}_0$.

Given two cuts, c over S and c' over S', we define $c \leq c'$ if and only if $\mathsf{S} \cap \mathsf{S}' \neq \emptyset$ and, for any process in the intersection, its history component in cut c is a prefix of its history component of in cut c'. Two cuts c and c' are *indistinguishable* to process p (written $c \sim_p c'$) when p's history is the same in both ($h_p = h'_p$).

2.1 Distributed, Coordinated Actions

We are concerned with applications in which specific events must occur uniformly (at all processes, if at any), and designate a special set of events, called *actions*. It is useful to think of these actions as requests to a service that is replicated for locality and/or fault tolerance. Any replica can receive and honor a request, but Uniformity means each request must eventually be 'logged' with the entire service. An action will arise unpredictably at a single process in any given run; while the set of actions can be described, the specific actions to appear in any run are not known *a priori*. If α arises at p in run ρ we say *p owns α* in ρ. The particular event whereby a process learns about the existence of an action does not have any externally observable effect on that process; it may occur anywhere in h_p provided it is before the first event that refers to α's existence (*e.g.*, p sending a message that mentions α).

In consequence, similar to the terminology of [10], the existence of an action is initially *local* to its owner. The intuition is that other processes learn about a non-local action by communicating with, or indirectly with, the action's owner. If α is an action, $do_p(\alpha)$ denotes the event at which process p does α.

[1] Histories of crashed processes can be made infinite by appending infinitely many *crash* events.

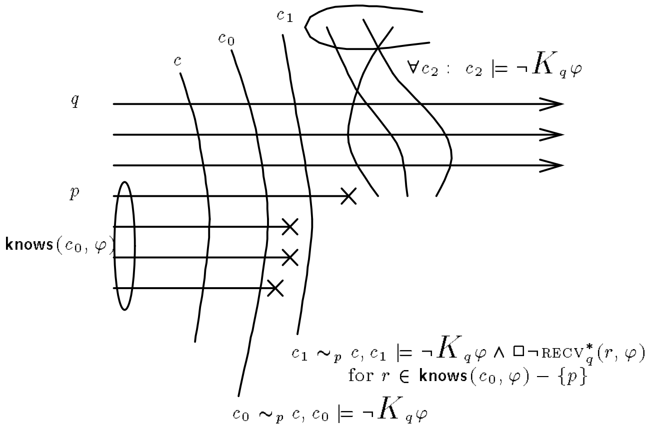

Figure 1: In the shown completions of c_1, process q does not learn φ.

2.2 The Formal Language

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

Time modalities $\Box\varphi$ (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, in *every* run that includes c, φ holds on *some* $c \leq c'$. In branching time logic, $\Box$ and $\Diamond$ are not duals. $\diamondminus\varphi$ (in the past) holds on c if and only if φ holds on some $c' \leq c$.

Knowledge modalities [14] $K_p\varphi$ (p knows) holds on c if and only if φ holds on all cuts $c' \sim_p c$. $S_{\mathsf{S}}\varphi$ (some process knows) holds on c if $c \models K_p\varphi$ for some $p \in \mathsf{S}$, while $E_{\mathsf{S}}\varphi$ (every process knows) holds on c if $c \models K_p\varphi$ for every $p \in \mathsf{S}$. $D_{\mathsf{S}}\varphi$ refers to distributed knowledge of φ within a set and holds if the knowledge of every $p \in \mathsf{S}$ together implies φ. For $p \in \mathsf{S}, E_{\mathsf{S}}\varphi \Rightarrow K_p\varphi \Rightarrow S_{\mathsf{S}}\varphi \Rightarrow D_{\mathsf{S}}\varphi$.

We refer to the earliest occasion at which $K_p\varphi$ holds, by the event $learn_p(\varphi)$. Crashed processes cannot 'know' anything; that is $K_p\varphi \Rightarrow \neg\mathrm{CRASHED}_p$.

Definition A formula φ is *ephemeral* to S, a set of processes, if it is possible that knowledge of it within S, can be irrevocably lost, $\diamondminus\Box\varphi \ \wedge\ \neg\Diamond D_{\mathsf{S}}\varphi$.

A formula φ is *persistent* in S if it always possible for a non-crashed member to learn it; that is, if it is always distributed knowledge within S, $\Box D_{\mathsf{S}}\varphi$.

Proposition 2.1 *In an asynchronous system in which a process cannot know with certainty that a particular message it sends will be received, then whenever φ is ephemeral to* S*, no $p \in \mathsf{S}$ can know only that another $q \in \mathsf{S}$ will eventually know φ :*

$$K_p \Diamond K_q \varphi \;\Leftrightarrow\; K_p K_q \varphi \;.$$

Proof : The $\Leftarrow$ direction follows trivially since $c \leq c$. We prove the $\Rightarrow$ direction by contradiction. Suppose $c \models K_p \Diamond K_q \varphi \wedge \neg K_p K_q \varphi$. Let $c_0 \sim_p c$ be a cut on which $\neg K_q \varphi$ holds. Let $\mathsf{knows}(c_0, \varphi)$ be the set of processes that know φ on c_0.

Now construct $c_1 \sim_p c_0 \sim_p c$, by crashing each $r \in \mathsf{knows}(c_0, \varphi)$ and 'dropping' every message in transit to q (Figure 1 illustrates). Let c_2 be identical to c_1 except p's own crash is its very next action. On c_2, all existing knowledge of φ is lost, and since φ is ephemeral, q may not learn it thereafter. That is $c_2 \models \neg \Diamond K_q \varphi$ and since $c_1 \leq c_2, c_1 \models \neg \Diamond K_q \varphi$. ∎

3 Generic Distributed Coordination

As in [21], the following formulas are used to define Distributed Coordination; they are all stable (*i.e.*, $\varphi \Rightarrow \Box \varphi$). Throughout the paper, we are interested in runs generated by solutions to the various Uniform Coordination problems we will define, not the entire set of asynchronous run of the system. Validity and truth should therefore be understood as pertaining only to these runs. We use $(*m)$ to refer to the contents of message m.

• $\textsc{recv}_p(q, m)$ holds on c if and only if $recv_p(q, m)$ is an event in c. $\textsc{recv}^*_p(q, m)$ holds when there is a sequence of processes $q = p_0, p_1, ..., p_k = p$ and messages, $m_0, m_1, ..., m_{k-1} = m$ such that $\textsc{recv}_{p_i}(p_{i-1}, m_{i-1})$ holds on c and for $0 \leq j \leq i, (*m_i) \Rightarrow (*m_0)$. We say m *acknowledges* m_0 and will use $\mathrm{ack}(m_0)$ to make this explicit. Similarly define $\textsc{send}_p(q, m)$ and $\textsc{send}^*_p(q, m)$.

• $\textsc{crashed}_p$ holds on c if $crash_p$ is an event in c.

• $\textsc{before}(e_1, e_2)$ holds on c if $e_1 \rightarrow e_2$ in c.

• $\textsc{did}_p(\alpha)$ holds on c if $do_p(\alpha)$ is an event in p's history component of c.

• 'α' holds on c when some non-crashed process knows the action α has arisen. While it is stable, 'α' is also ephemeral.

• $\textsc{owns}_p(\alpha)$ holds if and only if p is the process at which the action α originates (in a given run).

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 about by some process that is not crashed. The originator, or owner, of α is denoted p_α. Since actions arise at a single 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' : $\textsc{owns}_p(\alpha) \Leftrightarrow \textsc{before}(learn_p(\text{'}\alpha\text{'}), learn_q(\text{'}\alpha\text{'}))$ for all other q.

3.1 Uniform Coordination

The need for processes to first obtain permission for a uniform action is presented in detail [21] though we recap the main discussion here. Permission for actions arises because the exemption we use forces the application to make exempt processes to appear as if they were crashed. We call this a *terminal* exemption. With respect to Uniform Coordination, as crashed processes cannot take further actions, an exempt process must therefore be prevented from taking new actions.

A process is disabled when it reaches a point after which takes no further actions:

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

Lacking an omnipotent external entity, effecting $\text{DIS'D}(q)$ means participating processes need some way of revoking an exempt process's ability to take actions. In the final analysis, solutions to any problem that exerts punitive restrictions on exempt processes must incorporate revocation, and therefore permission. The formula $\text{PERMIT}_p(\alpha)$ holds when p has permission to take α.

Initially, no actions are permitted. Processes eventually take actions they know they have permission to take, $K_p\text{PERMIT}_p(\alpha) \Rightarrow \Diamond(\text{DID}_p(\alpha) \vee \text{CRASHED}_p)$, and attempt (or die trying) to get permission for actions they know about, $K_p\text{`}\alpha\text{'} \Rightarrow \Diamond(K_p\text{PERMIT}_p(\alpha) \vee \text{DIS'D}(p))$. Liveness precludes $\text{PERMIT}()$ from being identically false, by requiring that on each cut, at least some process that has started running is not disabled (the cut c defining $\mathsf{Started}()$, is the cut on which the formula is begin evaluated),

$$\exists p \in \mathsf{Started}() : \neg\text{DIS'D}(p)$$

The Uniformity clause says that all processes obliged to take an action either do so, or are eventually disabled:

$$K_p\text{PERMIT}_p(\alpha) \Rightarrow \bigwedge_{q\in\mathsf{S}_\alpha} \Diamond\Big(K_q\text{PERMIT}_q(\alpha) \vee \Diamond\text{DIS'D}(q)\Big) \;.$$

We restate the Accountability Theorem (Theorem 4.1) of [21] which proves that in any solution to Uniform Coordination a process knows it has permission to take an action if and only if it can account for every other obliged process with respect to that action,[2]

Accountability Theorem *If process p knows it has permission to take action α, then either p knows K_q'α', or p knows $\Diamond\text{DIS'D}(q)$ for every $q \in \mathsf{S}_\alpha$,*

$$K_p\text{PERMIT}_p(\alpha) \;\Leftrightarrow\; \bigwedge_{q\in\mathsf{S}_\alpha} K_pK_q\text{`}\alpha\text{'} \vee K_p\Diamond\text{DIS'D}(q) \;. \quad \blacksquare$$

[2]In the formulation of the theorem, $K_pK_q\text{`}\alpha\text{'}$ should, by rights, be $K_p\Diamond K_q\text{`}\alpha\text{'}$. As written, the formula implies p knows q is not crashed, which can never be the case.

Communication patterns in any solution are dictated by this result. First, knowing another process knows about an action involves communication, $K_p K_q$'α' $\Leftrightarrow$ $\text{RECV}^*_p(q, \text{ack}(\text{'}\alpha\text{'}))$. Second, Corollary 4.1 of [21] proves that knowing $\Diamond\text{DIS'D}(q)$ requires that p know a *majority* of q's future obligation sets will stop communicating with q, and that this, too, requires communication back to p. A process is said to be *shunning* another if has stopped communicating with it. Trivially, a crashed process shuns every other process. For notational convenience assume obligation sets are identical for all actions, and let M denote a majority subset of S.

Disabling Corollary *Before a process p can know another process q will eventually be disabled, p must know a majority of q's cohorts will eventually shun q,*

$$K_p \Diamond\text{DIS'D}(q) \Leftrightarrow \exists \mathsf{M} \subset \mathsf{S} \bigwedge_{r \in \mathsf{M}} \text{RECV}^*_p(r, \text{ack}(\diamond\text{SHUN}_r(q))) \ . \quad \blacksquare$$

This requirement percolates up to the propagation of α in that p can never hope to disable more than a minority of processes; if less than a majority acknowledge its announcement of α it can never hope to get permission for α.

We prove now that Uniform Coordination is equally hard when system channels are reliable, as when they drop messages. While surprising, this result is directly attributable to the terminal exemption, $\text{DIS'D}()$; this will not be the case for Uniform Coordination with a weaker exemption.

Theorem 3.1 *In an asynchronous system in which processes can crash, and either (i) channels may drop messages or (ii) processes use shunning to effect a terminal exemption, then no process knows with certainty that any given message it sends will be received.*

Proof : Case (i) is obvious. For Case (ii), we argue that a process must always consider the possibility that any other process (*i.e.*, any given destination) may be shunning it.

In the absence of specific conditions defining when one process attempts to disable another (though the usual trigger is the suspicion of faultiness) we only assume the genesis of such a desire is local to the process initiating it. Moreover, to ensure Liveness *a priori* 'treaties' with other processes are precluded, meaning q must always suspect such attempts. Finally, since clocks and communication are asynchronous, q can always imagine an attempt to disable it has been successfully propagated since the last message it received. $\blacksquare$

The implication of theorem 3.1 is that even when channels are reliable, a process participating in Uniform Coordination must wait for responses acknowledging its intention to take an action. Recall that the presence of terminal exemptions (*i.e.*, the need to make exempt processes appear as if they were crashed) gave rise to the need to revoke a process's ability to take actions, and thus to the need for a process to seek permission to take actions. Shunning essentially means a process cannot rely on messages that request permission being received, and this means channel reliability counts for nothing; Uniform Coordination is no easier when channels are reliable.

3.2 Sequenced Actions

Often, a subset of coordinated actions must be taken by all processes in some particular order. Canonical problems include linear and causal multicast, leader election, mutual exclusion, transactions, group membership changes, and so forth.

Definition A *Sequence Clause* requires that whenever two actions are related by some condition, all jointly-obliged processes take both actions in the same order,

$$\Sigma(\alpha, \beta) \Rightarrow \bigwedge_{p \in \mathsf{S}_\alpha \cap \mathsf{S}_\beta} \text{DID}_p(\beta) \Rightarrow \text{DID}_p(\alpha)\ . \tag{2}$$

The examples just given, and those will will consider, are all *dynamic* in that the ordering condition is defined relative to characteristics of each system run; static conditions (orderings between events that hold over all runs) require no discovery, and therefore no communication, on the part of participating processes.

3.2.1 Discussion

It may seem that the consequent in Formula 2 could be replaced by $\text{BEFORE}(do_p(\alpha), do_p(\beta))$. While it is implied by the current specification, the explicitly causal formulation would allow β to exist in a run without α (not the case in Formula 2). It is then important to consider whether one could predict, in any given run that α would never occur. That is, if $\Sigma(\alpha, \beta)$ holds and processes take β only, can it be guaranteed they will never be required to take α, violating the necessary ordering between the actions? This seems unlikely; moreover, if such a condition could be predicted, there would be no need for the sequence requirement in the first place. Alternatively, we could try to finesse the issue by having processes that have already taken β prevent any other process from taking α if it does arise. This is equivalent to allowing processes to ignore actions if they arise at 'inconvenient' times. As it violates the obligation processes have to take all actions that arise, it too is undesirable.[3]

There are more fundamental issues as well. Given that $\Sigma(\alpha, \beta)$ reflects dynamic conditions in a given run, it seems unnatural that the dependent action, β, would arise in a run without its trigger, α. Finally, an exclusionary clause, rather than sequence clause would be more appropriate to express "if β occurs in a run, then α cannot." A sequence clause is intended to model some notion of temporal consistency and dependency between actions; exclusion, in a sense, models anti-dependency. In consequence then, we use Formula 2 for the Sequence Clause and tie β's existence in a run with α's.

[3] One can imagine a service that ignores a particular client because the service itself acted precipitously.

3.2.2 Examples of Sequence Clauses

As an example clause, a partial order that respects causality may have $\Sigma(\alpha,\beta)$ reflect the causal relationship between the originating events for α and β. Let $Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha)$ denote the event by which p_α announces α to S_α.

$$\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \textsc{before}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta)) \ .$$

Instantiating $\textsc{did}_p(\alpha)$ with the p's delivery of message α and $Annc_{p_\alpha}(,\mathsf{S}_\alpha,\alpha)$ with the multicast of message α to process group S_α, gives the requirement for causal delivery order for multicast messages [6, 20, 18]. It is not hard to see that vector clocks [12] provide the necessary information.

A total order may have $\Sigma(\alpha,\beta)$ reflect some pre-ordained rule if α and β originate concurrently and adhere to causality otherwise. For example, if $\textsc{concurrent}(e,e')$ has the expected meaning for events e and e',

$$\begin{aligned}\Sigma(\alpha,\beta) &\stackrel{\text{def}}{=} \\ &\textsc{before}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta)) \ \vee \\ &\Big((p_\alpha < p_\beta) \ \wedge \ \textsc{concurrent}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta))\Big) \ .\end{aligned}$$

Such a definition is commonly found in attempts to linearize the partial 'happens before' order on events, multicast with linear delivery order [5, 18], and in serializability [3]. A more general total order that ignores causal origins defines $\Sigma(\alpha,\beta)$ as any process taking α before β,

$$\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \bigvee_{p\in\mathsf{S}_\alpha\cap\mathsf{S}_\beta} \textsc{before}(do_p(\alpha), do_p(\beta)) \ . \tag{3}$$

A sequence clause requiring any change to an action's obligation set to occur after the action itself can describe a generic notion of *virtual synchrony* [4] or, provided these changes are globally organized, Extended Virtual Synchrony [2].

$$\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \beta \in \Big\{\mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha \cup \{r\} \ , \ \mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha - \{q\}\Big\} \ .$$

3.3 Terminating Actions

A *Termination Clause* sets limits on when an action must be completed, in that all obliged processes have either taken the action or been deemed exempt from doing so. A common, practical example occurs in buffer flushing or garbage collection: a process will want to remove actions from a local buffer when it knows it no longer needs to store them for future reference. In simple Uniform Coordination, obliged processes will want to transfer outstanding actions to new processes joining the computation and, at a partition merge, those that have been on the other side of the partition; actions that have been terminated need not be transferred. Termination can also be considered a kind of timeliness guarantee for asynchronous systems: if some condition holds, then α has been done. In synchronous systems, such a condition is usually the global clock time; in asynchronous systems, broader global notions must suffice.

Definition A *Termination Clause* requires that if a certain condition holds with respect to an action, that action has been taken uniformly,

$$\tau(\alpha) \;\Rightarrow\; \bigwedge_{p \in \mathsf{S}_\alpha} \textsc{did}_p(\alpha) \vee \textsc{dis'd}(p)\;. \tag{4}$$

3.3.1 Discussion

Strong termination conditions are prevalent whenever a group of processes coordinate to implement a service whose behavior must be that of a single, fault-tolerant process. Intuitively, the previous coordinated action must be known to be terminated before before any replica can commit to a succeeding coordinated action. This applies to Strong Group Membership [23], and any of the family of resource allocation problems (*e.g.*, leader election, mutual exclusion, and token holder).

In theory, computers' finite storage capacity means termination is fundamental in any distributed coordination problem. Given the ability to store n actions, an initiating process need not attempt to disable obliged, yet unresponsive processes until its storage is exhausted. In that case, either the initiating process must halt, or it must take action to formally exempt the unresponsive processes and terminate the actions it is storing.

3.3.2 Examples of Termination Clauses

Simultaneous Termination requires an action to be taken at the same logical time (*i.e.*, on a consistent cut) by all processes,

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists p \in \mathsf{S}_\alpha : \textsc{did}_p(\alpha)\;.$$

Simple Uniform Coordination permits runs of limited divergence in which some errant processes may, for a finite period, ignore the behavior of the group. To preclude this, *Lock-Step Termination* requires any action a process has previously taken to to be terminated it before takes a next action,

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists \beta : \exists p \in \mathsf{S}_\alpha : \textsc{before}(learn_p(\textsc{permit}_p(\alpha)), learn_p(\textsc{permit}_p(\beta)))\;.$$

Lock-Step Termination still permits a process to take one divergent action (its last action). To preclude even this, we add the requirement that an action be terminated if some other process can concurrently take some other action,

$$\begin{aligned}
\tau(\alpha) \stackrel{\text{def}}{=}\; & \exists \beta : \exists p \in \mathsf{S}_\alpha : \\
& \textsc{before}(learn_p(\textsc{permit}_p(\alpha)), learn_q(\textsc{permit}_p(\beta))) \;\vee \\
& \exists \gamma : \exists q \in \mathsf{S}_\alpha \textsc{concurrent}(learn_p(\textsc{permit}_p(\alpha)), learn_q(\textsc{permit}_q(\gamma)))
\end{aligned}$$

Liveness then ensures no two processes take different i^{th} actions. The Strong Group Membership problem of [23] can be expressed as Uniform Coordination with Strong Lock-Step Termination (the coordinated actions are changes to some initial obligation set).

Perhaps most important, termination clauses define variations of virtual synchrony. For example, flush protocols tend to guarantee that all actions having the same obligation set are terminated before any action is initiated with a different obligation set,

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists\beta : \mathsf{S}_\alpha \neq \mathsf{S}_\beta \ \wedge \quad \text{BEFORE}(Annc_{p_\alpha}(\mathsf{S}_\alpha, \alpha), Annc_{p_\beta}(\mathsf{S}_\beta, \beta)) \ .$$

Termination with respect to a class of other system events $\mathcal{E}$, where e_p is p taking event e, is defined as

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists e \in \mathcal{E} : \exists p \in \mathsf{S}_\alpha : \text{BEFORE}(\text{DID}_p(\alpha), e_p) \ .$$

In [15], Hiltunen and Schlichting take $\mathcal{E}$ to be the set of messages delivered by a membership service to the application. With this interpretation, $\tau(\alpha)$ is the virtual synchrony provided in [6]. Now, requiring the events in $\mathcal{E}$ to adhere to other properties gives stronger termination clauses. Again building on [15], if the membership changes are themselves delivered in the same sequence at all site, the virtual synchrony obtained results in all obliged sites taking an action α during the same 'compositional moment' in the system's history; *i.e.*, between instances of change to the set of processes considered to be part of the system. When changes refer only to obligation sets (or, alternatively, when the set of processes defining the system is the obligation set), Extended Virtual Synchrony is obtained by sequencing membership changes, and by terminating α before any obliged process changes S_α,

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists e \in \Big\{\mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha \cup \{r\} \ , \ \mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha - \{q\}\Big\} : \exists p \in \mathsf{S}_\alpha : \text{BEFORE}(\text{DID}_p(\alpha), e_p) \ .$$

4 Learning and Propagating $\Sigma(\alpha, \beta)$

We concentrate now on communication requirements for solving Sequenced Uniform Coordination, and make only brief comments later for Terminated Uniformity. Through the rest of the paper, we assume all actions have the same obligation sets, denoting it S; the results can generally be extended to include overlapping obligations sets by quantifying over the intersection. We analyze how processes discover $\Sigma(\alpha, \beta)$ and, once discovered, under what conditions they can safely take β. The communication required to learn $\Sigma(\alpha, \beta)$ will depend on the degree to which it involves the local state of other processes. For example, if $\Sigma(\alpha, \beta)$ is local to process p, then p need never query any other q to learn it. On the other hand, to ensure safety and liveness, p should divulge $\Sigma(\alpha, \beta)$ to its cohorts. If p should crash (and $\Sigma(\alpha, \beta)$ is not persistent), its cohorts can never learn $\Sigma(\alpha, \beta)$; if they guess incorrectly, they violate safety, and if they make no guess, they must block forever. We use the various knowledge modalities to quantify these notions.

Any $\Sigma(\alpha,\beta)$ can be written as a function of each process's local state with respect to α and β. Any commonly known function or predicate on these local states can be used. If $\sigma_p(\alpha,\beta)$ is p's contribution to $\Sigma(\alpha,\beta)$, then we can write $\Sigma(\alpha,\beta)$ as

$$F\Big(\{\sigma_p(\alpha,\beta) \mid p \in \mathsf{S}\}\Big)\ .$$

In the centralized example above, if $\Sigma(\alpha,\beta)$ is local to p then $F()$ is commonly known to be $\sigma_p(\alpha,\beta)$. All processes wishing to learn $\Sigma(\alpha,\beta)$ must learn $\sigma_p(\alpha,\beta)$. In the case when $\Sigma(\alpha,\beta)$ is predetermined for every run and requires no discovery, then $\sigma_p(\alpha,\beta)$ is either $\top$ for all p or $\bot$ for all p, and $F()$ is commonly known to be the local component. In the more interesting cases, $\Sigma(\alpha,\beta)$ will be distributed knowledge among processes, and must be discovered.

Proposition 4.1 *In any solution to Sequenced Uniform Coordination, a process that is able to take two actions that must be sequenced, knows the order in which to take them,*

$$K_p\textsc{permit}_p(\alpha)\ \wedge\ K_p\textsc{permit}_p(\beta)\ \wedge\ \Sigma(\alpha,\beta)\ \Rightarrow K_p\Sigma(\alpha,\beta)\ .$$

Proof : Follows easily from safety of the sequence clause. ∎

Proposition 4.2 *In any solution to Sequenced Uniform Coordination, a process that knows about two actions that must be sequenced will, eventually, learn the required order or be disabled,*

$$K_p\text{`}\alpha\text{'}\ \wedge\ K_q\text{`}\beta\text{'}\ \wedge\ \Sigma(\alpha,\beta)\ \Rightarrow \Diamond\Big(K_p\Sigma(\alpha,\beta)\ \vee\ \textsc{dis'd}(p)\Big)\ .$$

Proof : Follows from the non-triviality requirement, logical reasoning, and Proposition 4.1

$$\begin{aligned}&K_p\text{`}\alpha\text{'}\ \wedge\ K_p\text{`}\beta\text{'}\ \wedge\ \Sigma(\alpha,\beta)\ \Rightarrow\\ &\Diamond\Big(\Big(K_p\textsc{permit}_p(\alpha)\ \wedge\ K_p\textsc{permit}_p(\beta)\Big)\ \vee\ \textsc{dis'd}(p)\Big)\ \wedge\ \Sigma(\alpha,\beta)\ \Rightarrow\\ &\Diamond\Big(K_p\Sigma(\alpha,\beta)\ \vee\ \textsc{dis'd}(p)\Big).\ \blacksquare\end{aligned}$$

Formula 5 applies knowledge-generalization to Proposition 4.1 to obtain,

$$K_pK_q\text{`}\alpha\text{'}\ \wedge\ K_pK_q\text{`}\alpha\text{'}\ \wedge\ K_p\Sigma(\alpha,\beta)\ \Rightarrow K_p\Diamond\Big(K_q\Sigma(\alpha,\beta)\ \vee\ \textsc{dis'd}(q)\Big)\ . \tag{5}$$

Proposition 4.3 *In any solution to Sequenced Uniform Coordination, a process that knows only about the second of two sequenced actions eventually learns the first (and their required order) or is disabled,*

$$\Sigma(\alpha,\beta)\ \wedge\ \neg K_p\Sigma(\alpha,\beta)\ \wedge\ K_p\text{`}\beta\text{'}\ \wedge\ \neg K_p\text{`}\alpha\text{'}\ \Rightarrow \Diamond\Big(\Big(K_p\text{`}\alpha\text{'}\ \wedge\ K_p\Sigma(\alpha,\beta)\Big)\ \vee\ \textsc{dis'd}(p)\Big)\ .$$

Proof : To maintain the sequence requirement, p can never be permitted to take β without having already taken α. A process knowing it has permission to take an action cannot be prevented from taking it, so $learn_p(`\alpha')$ (or at least $learn_p(\Sigma(\alpha,\beta))$) must occur before $learn_p(\text{PERMIT}_p(\beta))$. The result follows from non-triviality and $K_p`\beta'$. ∎

The next proposition is similar to the Accountability Theorem in that before a process can take an action, it must have decisive knowledge about its cohorts' knowledge of the action and any relative ordering for that action.

Proposition 4.4 *In any solution to Sequenced Uniform Coordination, if $\Sigma(\alpha,\beta)$ holds and p may take both α and β, then unless p knows $\Sigma(\alpha,\beta)$ is persistent, it knows a remote process also knows $\Sigma(\alpha,\beta)$, or it knows the remote process will be disabled,*

$$\Big(K_p\text{PERMIT}_p(\alpha)\ \wedge\ K_p\text{PERMIT}_p(\beta)\ \wedge\ \Sigma(\alpha,\beta)\Big)\ \Rightarrow K_pK_q\Sigma(\alpha,\beta)\ \vee\ K_p\Diamond\text{DIS'D}(q)\ .$$

Proof : From the Accountability Theorem, $K_p\text{PERMIT}_p(\alpha) \Rightarrow K_pK_q`\alpha'\ \vee\ K_p\Diamond\text{DIS'D}(q)$, and the same holds for β. Proposition 4.1 gives $K_p\Sigma(\alpha,\beta)$, so we obtain

$$\begin{aligned}
&\Big(K_p\text{PERMIT}_p(\alpha)\ \wedge\ K_p\text{PERMIT}_p(\beta)\ \wedge\ \Sigma(\alpha,\beta)\Big)\ \Rightarrow\\
&\Big(K_pK_q`\alpha'\ \vee K_p\Diamond\text{DIS'D}(q)\Big)\ \wedge\ \Big(K_pK_q`\beta'\ \vee K_p\Diamond\text{DIS'D}(q)\Big)\ \wedge\ K_p\Sigma(\alpha,\beta)\ \Leftrightarrow\\
&\Big(\Big(K_pK_q`\alpha'\ \wedge\ K_pK_q`\beta'\Big)\ \vee\ K_p\Diamond\text{DIS'D}(q)\Big)\ \wedge\ K_p\Sigma(\alpha,\beta)\ .
\end{aligned}$$

Rearranging, then applying Formula 5 gives

$$\begin{aligned}
&\Big(K_p\text{PERMIT}_p(\alpha)\ \wedge\ K_p\text{PERMIT}_p(\beta)\ \wedge\ \Sigma(\alpha,\beta)\Big)\ \Rightarrow\\
&K_p\Diamond\Big(K_q\Sigma(\alpha,\beta)\ \vee\text{DIS'D}(q)\Big)\ \vee\ K_p\Big(\Sigma(\alpha,\beta)\ \wedge\ \Diamond\text{DIS'D}(q)\Big)\ .
\end{aligned}$$

The proof now proceeds by contradiction. Suppose a cut c satisfies $\neg K_pK_q\Sigma(\alpha,\beta)\wedge\neg K_p\Diamond\text{DIS'D}(q)$. Then the right-hand disjunct of the consequent of Formula 4 cannot hold, leaving

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

Then there is a $\sim_p$-equivalent cut, c_1, satisfying $\neg K_q\Sigma(\alpha,\beta)$. By hypothesis p does not know whether $\Sigma(\alpha,\beta)$ is persistent, so there is a completion of c_1 throughout which q can never learn $\Sigma(\alpha,\beta)$. That is, $c_1 \models \neg\Diamond K_q\Sigma(\alpha,\beta)$.

The proof that c_1 also satisfies $\neg\Diamond\text{DIS'D}(q)$ is as in the Accountability Theorem. As the system has no omnipotent entity, it must be that some subset of S jointly control $\text{DIS'D}(q)$. That is, 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. In that case, nothing can prevent q from learning it has permission for new actions and the ability to disable q is lost. Thus, $\exists c_1 \sim_p c$, such that

$$c_1 \models \neg\Diamond K_q\Sigma(\alpha,\beta)\ \wedge \neg\Diamond\textsc{dis'd}(q) \text{ if and only if}$$
$$c_1 \models \neg\Big(\Diamond K_q\Sigma(\alpha,\beta)\ \vee \neg\Diamond\textsc{dis'd}(q)\Big) \text{ so}$$
$$c_1 \models \neg\Diamond\Big(K_q\Sigma(\alpha,\beta)\ \vee \neg\textsc{dis'd}(q)\Big)\ . \ \blacksquare$$

Corollary 4.1 *In any solution to Sequenced Uniform Coordination, a process can take the second of two ordered actions only when it can take the first, has learned the required ordering, and can guarantee all other processes will do likewise,*

$$K_p\textsc{permit}_p(\beta)\ \wedge\ \Sigma(\alpha,\beta)\ \Rightarrow$$
$$K_p\textsc{permit}_p(\alpha)\ \wedge\ K_p\Sigma(\alpha,\beta)\ \wedge\ \Big(K_pK_q\Sigma(\alpha,\beta)\ \vee\ K_p\Diamond\textsc{dis'd}(q)\Big)\ .$$

Proof : Follows from the Sequence clause and Proposition 4.4. $\blacksquare$

4.1 Communication Costs for Sequenced Uniform Coordination

Corollary 4.1 tells us that in Sequenced Uniform Coordination, the cost of doing action β is the cost of doing α, the cost of learning $\Sigma(\alpha,\beta)$, and the cost of being able to account for remote processes with respect to $\Sigma(\alpha,\beta)$. Of these three, the cost of learning $\Sigma(\alpha,\beta)$ is what distinguishes one variant of Sequenced Uniform Coordination from another. Our modular specification, our description of $\Sigma(\alpha,\beta)$ as a function of processes' local states, and the Chandy-Misra results [10] make this cost readily computable; whenever p must know something about q's local state (for example, K_q'α', or $\sigma_q(\alpha,\beta)$), p must receive a message from q implying the needed information; in general, p must actively request that information. In the absence of q's response, p must attempt to disable q. The Disabling Corollary proves this involves communication between p and a majority subset of S. Protocol designers will want to analyze whether the total cost of each process learning $\Sigma(\alpha,\beta)$ (or, for that matter, $\Diamond\textsc{dis'd}(q)$ or K_q'α') is greater than the cost of one process learning it and disseminating it.

Notice that a process incurs the cost of accounting for remote processes' knowledge of $\Sigma(\alpha,\beta)$ only when it does not know if $\Sigma(\alpha,\beta)$ is persistent. Intuitively, the worry is that p learns $\Sigma(\alpha,\beta)$, acts according to it, then immediately crashes possibly taking all knowledge of $\Sigma(\alpha,\beta)$ with it. As p cannot guarantee it will survive long enough to pass on its knowledge of $\Sigma(\alpha,\beta)$, it must either propagate its knowledge before taking β or ascertain other processes' knowledge. Dwork and Skeen touch on this issue when determining how early a process can commit a decision [11]; in their non-blocking protocol, a process can commit a decision as soon as (*i.e.*, in the first round in which) it knows it will not need to respond to any future queries about the tentative decision. The problem of 'invisible' commits in asynchronous systems is also discussed in [23].

We have not assumed predetermined bounds on the number of failures, so persistence effectively means total propagation. However, it is common to see solutions to distributed coordination parameterized by the number of failures they can withstand. This is the notion of t-fault-tolerant solutions. For the cost analysis of Sequenced Uniform Coordination, a process would know $\Sigma(\alpha, \beta)$ were persistent if it knew at least $t+1$ processes knew it.

This makes sense for Sequenced Uniform Coordination even though we do not assume an upper limit on the number of failures the system may experience. Specifically, since effecting the terminal exemption $\text{DIS'D}(q)$ requires a majority of processes to stop communicating with q, solutions to Uniform Coordination (and therefore Sequenced Uniform Coordination) cannot ever disable more than a minority of participants. Since no solution can make progress without a majority of processes, it is sensible to declare a formula persistent if at least a majority of processes know it.

5 Learning and Propagating $\tau(\alpha)$

The statements from the previous section apply to learning and propagating $\tau(\alpha)$. For example, $\tau(\alpha)$ can, like $\Sigma(\alpha, \beta)$, be expressed as a function of processes' local states, thus illuminating the communication required to learn it. In contrast to $\Sigma(\alpha, \beta)$, it can be argued that it is not vital for $\tau(\alpha)$ to be persistent. Certainly, the Termination condition itself must be maintained for safety, but the consequences of the system losing knowledge of $\tau(\alpha)$ do not affect the overall agreement on actions taken, or result in any significant conflicts between actions. Instead, termination in its most abstract form models timeliness or efficiency guarantees. For example, in the case of garbage collecting or buffer flushing, the inability of a process to learn $\tau(\alpha)$ results only in a process not being able to purge α from a local storage buffer. While this is certainly an inefficiency, it does not affect a protocol's overall correctness.

Ideally, the condition $\tau(\alpha)$ should be locally observable to a process; if it requires discovery of the full global state, the process would do equally well to detect termination itself as $\tau(\alpha)$. In the terminology of Taylor and Panangaden [19], it may be desirable for $\tau(\alpha)$ to hold on a locally-distinguishable consistent cut, meaning $\tau(\alpha)$ would be concurrent common knowledge. This corresponds to Simultaneous Termination; whether it is achievable when processes can crash is not known.

6 Conclusions

We presented a broad characterization of distributed coordination, defining a generic sequence clause and a generic termination (or timeliness) clause for the actions being coordinated. We demonstrated how these, layered on top of the Uniform agreement requirement, can be instantiated to describe many problems common in the literature. By deconstructing distributed coordination in this way, we factored out the cost of coordination and isolated costs due solely to ordering and termination. Specifically, we gave an absolute way to compare instances of Sequenced Uniform Coordination. Our works also builds on a single platform, a specific asynchronous system and a simple coordination problem. Our results can therefore consistently compare distributed coordination problems. This, coupled with our modular decomposition of coordination, is important as toolkits for distributed applications programming evolve and give users more flexibility in choosing (rather than imposing) communication and process group semantics.

We also showed that any problem incorporating terminal exemptions (of which Uniform Coordination is an example) is no easier when communication channels are reliable than when they are lossy. While this result implies a severe limitation on what is possible in networks that are prone to partition, it also pinpoints exactly why the limitation arises. We are currently investigating less stringent 'punishments' for processes exempt from taking some of the coordinated actions.

It is relevant to question whether the examples given in Sections 3.2.2 and 3.3.2 reflect the intentions of the problems' designers; in some cases this will be impossible to answer. However, the exercise in attempting to provide formal semantics for the sequences and synchrony actually provided is nonetheless useful. As well, minor modifications can bring new, appropriate notions to light. Given the required information flow, we are applying these results to existing protocols (an approach similar to [8]) to examine both correctness and efficiency. Certainly vector clocks suffice to carry information about actions' causal origins, but conditions for other orderings are more complex. Even for causal orderings, termination will describe when it is safe to prune piggybacked information to keep messages small.

The issue of processes controlling each others' destiny needs to be studied in more detail. The notion employed here arises from our previous results, which in turn have been influenced by our work in Isis. However, in more loosely structured systems, such strong control (strong punitive reaction for exempting a process) may be inappropriate.

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] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *Proceedings of the 13th IEEE ICDCS*, pages 551–560, May 1993.

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

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

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

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

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

[8] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

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

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

[11] C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commitment. In *Second PODC*, pages 1–11. ACM, August 1983.

[12] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, 1988.

[13] 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.

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

[15] M. Hiltunen and R. Schlichting. Properties of Membership Services. In *Second Int'l Symposium on Autonomous Decentralized Systems*, 1995. To appear.

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

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

[18] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel and Distributed Systems*, 1990.

[19] P. Panangaden and K. Taylor. Concurrent Common Knowledge: Defining Agreement for Assynchronous Systems. *Distributed Computing*, 6(2):73–94, 1992.

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

[21] A. Ricciardi. Dissecting Distributed Coordination in Asynchronous Systems. Technical Report ECE-PDS-9502, The Univerisity of Texas, 1995.

[22] A. Ricciardi. Process Exemptions in Distributed Coordination. Technical Report (in preparation), The Univerisity of Texas, 1995.

[23] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Tenth PODC*, pages 341–351. ACM, 1991. (also Cornell University TR93-1328).

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

[25] 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.