Uniform Actions in Asynchronous Distributed Systems

Dalia Malki* Ken Birman† Aleta Ricciardi‡ André Schiper §

TR 94-1447

Department of Computer Science, Cornell University

Abstract

We develop necessary conditions for the development of asynchronous distributed software that will perform *uniform* actions (events that if performed by any process, must be performed at all processes). The paper focuses on *dynamic uniformity*, which differs from the classical problems in that processes continually leave and join the ongoing computation. Here, we first treat a static version of the problem (lacking joins), and then extend the results so obtained to also include joins. Our results demonstrate that in contrast to Consensus, which cannot be solved in asynchronous systems with even a single faulty process, dynamic uniformity can be solved using a failure detection mechanism that makes bounded numbers of mistakes. Because dynamic uniformity arises in systems that maintain safety within a "primary partition" of a network, our paper provides a rigorous characterization of the framework upon which several existing distributed programming environments are based.

1 Introduction

Our work on fault-tolerant distributed systems (Isis [5], Transis [1, 14], and Horus [20]) gives rise to a desire to understand the theoretical foundations upon which such systems depend. These systems all use the virtual synchrony programming model [3], so it may not be surprising that they employ similar lower-level mechanisms. Nonetheless, we are not aware of *any* asynchronous distributed system that uses a different low-level architecture and also provides the same environment: dynamic addition and removal of processes, and

*Institute of Computer Science, The Hebrew University of Jerusalem, Israel. Supported in part by GIF I-207-199.6/91.

†Computer Sci. Dept., Cornell University, Ithaca, NY, USA. Supported by ARPA/ONR grant number N00014-92-J1866.

‡Electrical & Computer Eng., U.Texas at Austin, USA

§Ecole Polytechnique Fédérale, Lausanne, Switzerland. Supported by the "Fonds national suisse" and OFES contract number 21-32210.91, as part of the ESPRIT Basic Research Project Number 6360 (BROADCAST).

non-trivial consistency properties. This observation led us to speculate that asynchronous distributed systems might be subject to conditions that dictate the lowest levels of the system architecture. If one can identify necessary conditions for the solution of some basic problems, these conditions will also define the software architecture for a wide range of real-world distributed systems.

We believe that *Dynamic Uniformity* (D-Uniformity) is the fundamental property required for many applications in asynchronous settings. Recently, focus has been drawn to uniform, or consistent, coordination, a strengthening of traditional agreement problems [15, 2, 12]. In non-uniform agreement, faulty processes are never required to behave consistently with their cohorts, even before they commit a fault. Uniform agreement addresses this by requiring that decisions made by any process in the system, even one that is later declared faulty, must eventually be made by every other active process; only failure and forcible removal (forced inactivity) excuse a process from complying.[1] D-Uniformity applies uniform agreement to systems in which the set of processes that are agreeing is continually changing, and in which uniform agreement is repeatedly being solved.

A further pragmatic motivation for considering D-Uniformity is that in a large distributed system, processes will often act on behalf of the system as a whole. While actions may occur locally, they must eventually be known to the entire membership. D-Uniformity captures the required propagation of such actions, as well as the obligation of other processes to take these actions, while avoiding notions such as 'correct/incorrect' processes.

D-Uniformity is trivial to solve if communication channels are reliable and the membership of the system is known to all process. When channels are lossy, a process initiating an action must know that its cohorts are aware of the action it is initiating. Our first result shows that D-Uniformity is achievable provided that a majority of the processes do not crash. Thus, dynamic uniformity is a problem for which coordinating the activity of only a majority of the processes suffices, and is therefore strictly weaker than *Distributed Consensus* [16].

We then explore the effect of adding a *membership service* to our asynchronous environment. We assume that a membership service reports failures, and possibly joins, to each process.[2] We show that, given a system of size N and a membership service that can make an infinite number of mistakes (*i.e.*, reporting a functional process as failed), but only $t-1$ mistakes at each process event, D-Uniformity can be solved provided there are at most $N-t$ failures. Thus, our work explains how distributed systems can be fault-tolerant "despite" the result of Fischer, Lynch and Patterson [11]: they can do so provided that the failure detection mechanism employed is sufficiently accurate (though it need not be as accurate as the failure detector needed to solve Consensus).

In much prior work, Consensus has been considered the basic problem in achieving distributed coordination [16]. For example, Chandra and Toueg show that Consensus is equivalent to both fault detection and to atomic broadcast [7]. It is well known that Consensus has no solution in asynchronous environments in which even a single process may crash [11].

[1] Systems like Isis, Transis and Horus are unable to detect failures accurately, so unresponsive processes are excluded from participation in the system, as if they had crashed. If communication to such a process is reestablished, it rejoins the system under a new process identifier.

[2] While similar in role to a *failure detector* [7], a membership service reports several types of membership events (*e.g.*, joins, and leaves) in addition to suspected failures.

In contrast, our work shows D-Uniformity is strictly weaker than Consensus when all but two processes may crash, and as hard as Consensus only when all but one process may crash. As a result, Consensus *per se* is not the basic problem underlying all dynamic coordination in asynchronous environments.

Finally, we address systems in which the set of processes that participate is itself dynamically defined; indeed, modeling the behavior of a system that supports both join events and failures is itself a non-trivial contribution of the paper. We show that D-Uniformity can be solved continuously, in the sense that coordinated actions need not be blocked to accommodate new processes. Moreover, process additions need not occur in the same order at participating processes, so that D-Uniformity can get by with a very weak process membership service. To solve D-Uniformity in this setting it suffices to keep a majority of each established system configuration alive. This is of interest because D-Uniformity underlies solutions to the "primary partition" membership problem, which arises in many distributed systems.

In the systems we study, D-Uniformity characterizes the lower levels of the system. However, these systems also provide higher level services, such as *totally-ordered* activity and *terminated* actions. Unlike D-Uniformity, these problems are evidently equivalent in difficulty to the Consensus problem. Generally, therefore, the conditions under which these systems guarantee progress in all layers may still be subject to the basic impossibility result of Fischer, Lynch and Patterson [11]. The analysis provided here gives the necessary and sufficient conditions for the lower level services only. Whether these services, by themselves, are of use for solving "interesting" coordination problems, is something yet to be studied.

2 System Model

The system consists of a finite set S of process identifiers. In this and the next two sections, we simplify the discussion by assuming that the system boots with all processes in S running; in Section 5, we will relax this assumption to more accurately model systems in which processes join computations in progress. Processes communicate with each other by passing messages. 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 always follows its assigned protocol.[3]

Messages between processes may be lost, but to compensate for this, processes re-send messages until acknowledged. Specifically, if processes p and q both remain operational, each message from p will eventually reach q; if p crashes, messages in transit may never be delivered. There are no permanent partitions.[4] Finally, we assume the network does not corrupt messages.

Let $\mathcal{A}$ refer to this asynchronous environment. Since messages may be delayed arbitrarily long, no process can determine whether an unresponsive process has crashed, or whether it

[3] The $crash_p$ event is not performed by p itself, nor is it visible to other machines within the system; it is introduced to simplify the discussion.

[4] The results presented here extend to the general omission failure model with only minor changes, provided the communication channels are FIFO. However, in this model, every process that omits sending or receiving messages needs to be counted as faulty.

only appears to have crashed.

It is natural to model processes as I/O state automata: A process has a local state and a transition function. We model the change in a process' local state with an *event* in the execution of the process. A history for process p, h_p, is a sequence of events beginning with the unique event $start_p$: $h_p = start_p \cdot e_p^1 \cdots e_p^k$, $k \geq 0$. A *cut* is a tuple of finite process histories, one for each $p \in \mathsf{S}$. The initial system cut, c_0, consists only of all the $start_p$ events (this will be discussed again in Section 5 to accommodate processes joining later in the computation). We assume familiarity with inter-event causality and the "happens-before" relation ($e \rightarrow e'$) [13], and with consistent cuts [8]. A system *run* is an n-tuple of infinite process histories, one for each process in S.[5] A cut, then, is a finite prefix of a run.

Actions

Since we are concerned with ensuring that specific events occur at all processes if at any, we designate a special set of events, called *actions*. While the set of actions can be described, the specific actions to appear in any execution are not known *a priori*. Actions will arise at one process, which we model by saying that each action is *owned* by one process. In the terminology of Chandy and Misra [9], the identity of an action is initially *local* to its owner, meaning that processes only learn about non-local actions by communicating with (or indirectly with) the action's owner. If β is an action, let $does_p(\beta)$ denote the event whereby process p performs β. While realistic instances of the uniformity problem would have enabling conditions associated with each action, for our purposes all actions are enabled all the time.

The Formal Language

We use a logic with temporal and knowledge modalities to define system properties. The basic formulas are propositional, and all formulas are evaluated along consistent cuts. When formula φ holds on cut c, we write $c \models \varphi$. The modalities have the following semantics:

- $\Box\varphi$ (always) holds on c if and only if φ holds on c and on all completions of c in any run,
- $\Diamond\varphi$ (eventually) holds on c if and only if, in *every* run that includes c, φ holds on *some* future cut,
- $K_p\varphi$ (p knows) holds on c if and only if φ holds on all cuts in all runs, in which p's local state is identical to its local state at c.

3 Dynamic Uniformity

The basic propositional formulas of the D-Uniformity problem are:

- $\text{DID}_p(\beta)$ holds on c if and only if $does_p(\beta)$ is an event in p's history component of c,

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

- $\textsc{owns}_p(\beta)$ if and only if p is the owner of β,
- K_p"(β)" holds exactly when p knows 'about' the action β. Obviously, $\textsc{owns}_p(\beta) \Rightarrow K_p$"$(\beta)$", but K_p"(β)" $\wedge \neg\textsc{owns}_p(\beta)$ holds only after p received a message naming β.
- $\textsc{crash}_p$ holds on c if and only if $crash_p$ is an event in c, and
- $\textsc{before}(e_1, e_2)$ holds on c if and only if $e_1 \to e_2$ in c.

The safety condition for D-Uniformity states that if any process in S takes an action, then every other process eventually does so as well or becomes disabled (we discuss this below in more detail):

$$p \in \mathsf{S} \wedge \textsc{did}_p(\beta) \Rightarrow \bigwedge_{q \in \mathsf{S}} \Diamond\Big(\textsc{did}_q(\beta) \vee \textsc{disabled}_q\Big) \tag{1}$$

Knowledge of an action forces a process to try to execute it: K_p"(β)" $\Rightarrow \Diamond\textsc{did}_p(\beta) \vee \Diamond\textsc{disabled}_p$.

To define $\textsc{disabled}_q$ precisely, we introduce the notion of *permission* to execute an action. The need for permission arises from the network assumptions, which force a process intending to execute an action to make its intention known, and to know that its intention is known. The formula $\textsc{permit}_p(\mathsf{S}, \beta)$ holds if p has permission within S to execute β. While we do not specify what constitutes permission (*e.g.*, whether it is granted only by a designated process, or by a quorum subset), we do require it to satisfy:

1. No process acts without permission
   $\textsc{did}_p(\beta) \Rightarrow \textsc{permit}_p(\mathsf{S}, \beta)$.
2. Initially no actions are permitted
   $c_0 \models \forall\beta \left(\bigwedge_p \neg\textsc{permit}_p(\mathsf{S}, \beta)\right)$.
3. Permitted actions are eventually executed if a process does not crash
   $\textsc{permit}_p(\mathsf{S}, \beta) \Rightarrow \Diamond\Big(\textsc{did}_p(\beta) \vee \textsc{crash}_p\Big)$.

Now, $\textsc{disabled}_q$ holds exactly when q will never get permission to execute actions for which it does not already have permission:

$$\textsc{disabled}_q \stackrel{\text{def}}{=} \Big(\neg\textsc{permit}_q(\mathsf{S}, \beta) \Rightarrow \Box\neg\textsc{permit}_q(\mathsf{S}, \beta)\Big) .$$

Formula (1) does not require $\textsc{did}_q(\beta)$ or $\textsc{disabled}_q$ to hold before p can execute $does_p(\beta)$.

We define a liveness condition to preclude trivial solutions in which all processes are disabled after a finite number of actions: We say that a system cut is live if some process p can execute at least one more action. Liveness for D-Uniformity requires all (consistent) cuts in all runs satisfying (1) to be live, *i.e.*$\bigvee_{p \in \mathsf{S}} \neg\textsc{disabled}_p$. An immediate consequence is that a process that is never disabled can eventually obtain permission to execute an infinite number of its actions.

|| **endless loop:** /* *of process* p */
    select β fairly out of the following:
        1. β is a new action ;
        2. β is removed from *pending-buffer* ;

    /* *Initiate* β */
    send(*announce*(β)) to all the processes in S ;
    wait for $\lceil N/2 \rceil$ ack(β) s ;
    $does_p(\beta)$;

|| **upon receive(***announce*(β)**) from** q**:**
    send(ack(β)) to q ;
    if β received for the first time
        store β in *pending-buffer* ;

Figure 1: A solution to D-Uniformity that is resilient to $\lceil N/2 \rceil$-1 failures.

3.1 When is D-Uniformity Solvable?

D-Uniformity is solvable provided only a minority subset of S crash. A corollary to this shows that D-Uniformity is strictly weaker than consensus by giving a simple reduction from a solution to consensus to a solution to D-Uniformity.

Lemma 3.1 *Let* S *be a system of* N *processes, such that each process owns an infinite number of actions. If fewer than* $\lceil N/2 \rceil$ *failures occur throughout the execution, then D-Uniformity is solvable.*

Proof: The protocol in Figure 1 solves D-Uniformity (the symbol || denotes concurrent threads). There, "fairly" means no action, owned or buffered, is never selected. The protocol is live because each non-faulty process succeeds in initiating an action within finite time. It is safe because for each performed action β, a message announcing β reaches at least one process that never crashes, guaranteeing that the action will eventually propagate to every other live process. Thus, every non-disabled process learns about every action in the system, eventually succeeds in performing it, and therefore executes every action in the system. ∎

Corollary 3.2 *D-Uniformity is strictly weaker than distributed consensus.*

Proof: Lemma 3.1 shows that D-Uniformity has solutions in an environment in which the consensus problem is unsolvable [11]. To see that D-Uniformity is solvable whenever consensus is solvable, assume CONSENSUS(*val*) is a multi-valued consensus atom.[6] That is, at each correct process, the consensus atom returns, in finite time, the same value.
Now each process can repeatedly perform the following to solve D-Uniformity:

[6]It is well known that, unless the system is prone to byzantine failures, multi-valued consensus can be implemented given binary-consensus.

choose a new action β;
$res := \textsc{consensus}(\beta)$;
perform res;

While res will often differ from β (*i.e.*, a process often executes a different action from the one it was asking for), since CONSENSUS() does not predetermine results, β will eventually be selected. Further, no safety conditions are violated since actions do not conflict in D-Uniformity. ∎

Lemma 3.3 *If $\mathcal{U}$ is an r-resilient solution to D-Uniformity, then $r < \lceil N/2 \rceil$.*

Proof: The proof is by contradiction. Intuitively, the idea is to partition S into S_1 and S_2 such that both S_1 and S_2 are independently viable. After that, delay communication between the two until the two sets begin operating independently of each other. We then crash all of the processes in S_2, leaving S_1 viable, but unable to recover information about the last actions performed in S_2.

Formally, assume $\mathcal{U}$ is an r-resilient solution to D-Uniformity, for $r \geq \lceil N/2 \rceil$. Let S_1 and S_2 partition S with $|\mathsf{S}_1| = r$. Let ρ be a run of $\mathcal{U}$ in which all processes in S_1 crash at some cut c. Because ρ is live, there must be a cut c', also in ρ and after c, as well as an action β, owned by a member of S_2 and unknown to S_1, such that $c' \models \textsc{did}_p(\beta)$, for $p \in \mathsf{S}_2$.

Now let ρ' be another run of $\mathcal{U}$ with the same prefix c, except that in ρ' all messages to and from S_2 are delayed after c. We claim that c' from above is also a cut in ρ'. To see this, observe that the two runs are indistinguishable to the processes in S_2, and that these processes will therefore take the same actions in both runs up to cut c'. Specifically, action β is also performed in ρ' with no communication involving any process in S_1.

Now, at c' in ρ', crash all processes in S_2. By assumption, $|\mathsf{S}_2| < r$, so in this execution fewer than r processes have crashed by cut c'. Thus, ρ' is still live in the sense that there is some $q \in \mathsf{S}_1$ that continues performing an infinite number of its actions. However, in our model, no message informing q about β need ever reach q, precluding q from ever performing β. This violates safety, and so $\mathcal{U}$ cannot be a solution to D-Uniformity. ∎

4 Increasing Resiliency with a Membership Service

Section 3 showed that D-Uniformity is solvable whenever a majority of the system is operational. In this section, we extend the asynchronous environment $\mathcal{A}$ to increase the resiliency of solutions to D-Uniformity. Specifically we augment $\mathcal{A}$ with a *membership service.* A membership service reports to every process p, at every event e in p's history, a *local view* $\mathsf{LocalView}_p(e)$, which is a list of process identifiers. We simply use $\mathsf{LocalView}_p$ when the event is clear.

4.1 Using a Membership Service

To give some intuition on how to use a membership service, recall the protocol of Figure 1. We might modify that protocol so process p awaits acknowledgments (in line 7) only from processes in $\mathsf{LocalView}_p$. However, a naive membership service could lead to undesirable results. For example, a membership service based on timeout would remove from $\mathsf{LocalView}_p$ any process that does not respond to p's messages within some time bound. Without further constraints, this membership service is unsafe for the purposes of D-Uniformity, even in failure-free runs.

Another example membership service might maintain agreement about suspected failures (*e.g.*, the service described in [17]). The advantage of such a service is that the protocol in Figure 1 can make safe progress even when $\lceil N/2 \rceil$ or more of the processes have failed. Informally, this is done by successively reconfiguring the system into decreasing majority sets. However, this approach requires agreement among a majority of the processes, and an inopportune combination of mistaken failure suspicions and true crashes between successive reconfigurations will cause the system to block[7].

4.2 The Weakest Membership Service Solving D-Uniformity

Our primary interest is to define the weakest membership service that will permit solutions to D-Uniformity in the presence of $\lceil N/2 \rceil$ failures or more. We use the definition from [6] to compare membership services.

Definition 1 (Chandra & Toueg) Let MS and MS$'$ be membership services. MS is *weaker than* MS$'$ if there is a protocol that, using MS$'$, can implement MS.

In proving the requisite properties of a weakest membership service for D-Uniformity, we make no assumptions on the values it provides, or on how it is used by the protocol.

Let $\mathcal{U}$(MS) ($\mathcal{U}$ operating in the presence of membership service MS) be a solution to D-Uniformity that is resilient to f failures, for $f \geq \lceil N/2 \rceil$. Let $t = N - f$ be the number of processes that do not crash. Lemma 4.1 describes a cut that cannot occur in any execution of $\mathcal{U}$(MS), while Lemma 4.2 shows that waiting for 'enough' acknowledgments (direct or indirect) before executing any action is the only way to prevent such a cut.

Lemma 4.1 *Let $\mathcal{U}$(MS) be a solution to D-Uniformity that is resilient to f failures. Let $t = N - f$, and let $\{p, q_1, \ldots, q_t\} \subseteq \mathsf{S}$. Then in no execution of $\mathcal{U}$(MS) is there a consistent cut c such that*

$$c \models \text{DID}_p(\beta) \bigwedge_{1 \leq i \leq t} \left(\neg\text{CRASH}_{q_i} \wedge \neg K_{q_i} \text{“}(\beta)\text{”} \right)$$

Proof: Assume to the contrary that c can occur in some execution of $\mathcal{U}$(MS). At c, suppose an adversary crashes all processes except $q_1, \ldots, q_t$. By assumption, the system remains live, and so one of the q_i continues performing actions indefinitely. In our model, this q_i may never learn about β, violating safety. ∎

[7] For example, in a system of five processes, the false removal of two of them may be fatal, for the real failure of two of the remaining three processes will block the system

Definition 2 Let p perform an action β. Let $\mathsf{acks\text{-}rcvd}_p(\beta)$ be the set of processes from which p received direct or indirect acknowledgment regarding β up to the event $does_p(\beta)$.

$$\mathsf{acks\text{-}rcvd}_p(\beta) \stackrel{\text{def}}{=} \left\{ q \mid K_p K_q \text{``}(\beta)\text{''} \right\}$$

Lemma 4.2 *If $\mathcal{U}(MS)$ is an f-resilient solution to D-Uniformity, and* $\text{does}_p(\beta)$ *occurs in any run, then at least one process in* $\mathsf{acks\text{-}rcvd}_p(\beta)$ *never crashes.*

Proof: Suppose to the contrary that by some cut c, every process (except p) in $\mathsf{acks\text{-}rcvd}_p(\beta)$ has crashed. Since at least t processes never crash, there is a set of processes $\{q_1, \dots q_t\} \subseteq (S \setminus \mathsf{acks\text{-}rcvd}_p(\beta))$, such that

$$c \models \text{DID}_p(\beta) \bigwedge_{q_i} \Big(\neg\text{CRASH}_{q_i} \wedge \neg K_p K_{q_i} \text{``}(\beta)\text{''} \Big).$$

Let c' be a cut that is p-equivalent to c, and satisfies $c' \models \neg K_{q_i} \text{``}(\beta)\text{''}$. Now define $\hat{c}$ to be identical to c' except that p's history component in $\hat{c}$ terminates with the event $crash_p$. This cut violates Lemma 4.1. (Figure 2 depicts such a bad scenario, with $t = 1$). ∎

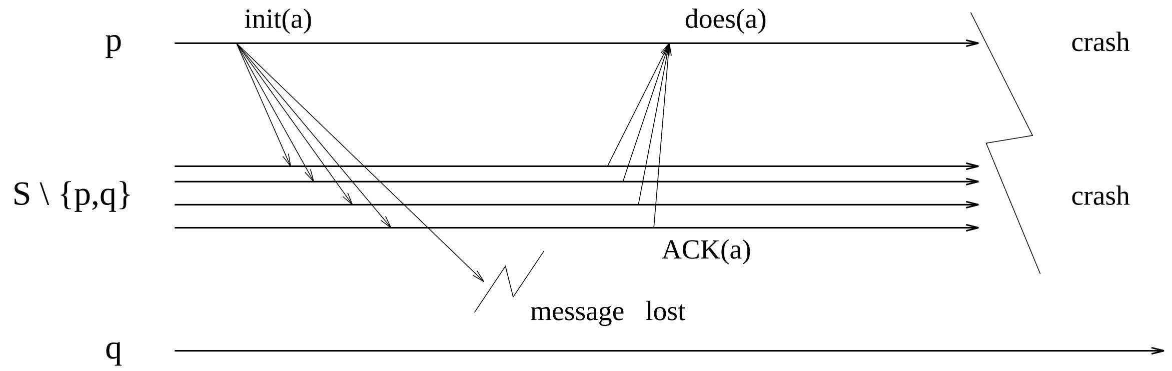

Figure 2: Bad scenario occurring if Lemma 4.2 is violated

As in [7] a membership service makes a mistake if and only if it removes a non-faulty process from the local view of some process.

Definition 3 A *Weak Membership Service that makes m* mistakes (WMS(m)) provides each process p at each event e, with a local view $\mathsf{LocalView}_p(e)$ such that:

1. The initial view of all processes is identical: $\mathsf{LocalView}_p(start_p) = \mathsf{LocalView}_q(start_q)$ for every $p, q \in \mathsf{S}$.

2. Crashed processes are eventually removed from the local view of each active process that remains active:

$$\text{CRASH}_q \Rightarrow \bigwedge_{p \in \mathsf{S}} \Diamond \Big(q \notin \mathsf{LocalView}_p \;\vee\; \text{DISABLED}_p \Big)$$

3. At most m non-crashed processes are mistakenly removed from $\mathsf{LocalView}_p$. We further distinguish *stable* (*i.e.*, permanent) removal from *non-stable* removal:

   Stable-removals: If removals are stable, then WMS(m) does not make more than m accumulated mistakes for each process. Thus, whenever $m+1$ processes are removed from $\mathsf{LocalView}_p$, at least one of these is, in fact, crashed.

   Non-Stable removals: If removed processes may be returned to local views, then at any event e there are at most m processes currently removed from $\mathsf{LocalView}_p(e)$ that are not crashed.

We now show that that a WMS($(N-f)-1$) membership service (*i.e.*, WMS($t-1$)) is the weakest for which f-resilient solutions to D-Uniformity exist. Necessity and sufficiency are shown in the next Theorems.

Theorem 4.3 *WMS($t-1$) is weaker than any membership service extending the asynchronous environment $\mathcal{A}$ in which D-Uniformity can be solved with f failures.*

Proof: Consider the following membership service: Take $\mathsf{LocalView}_p(start_p)$ to be S; at each event $does_p(\beta)$, set $\mathsf{LocalView}_p$ to $\mathsf{acks\text{-}rcvd}_p(\beta)$, but otherwise, do not change $\mathsf{LocalView}_p$.

We show this membership service belongs to the class of WMS($t-1$) membership services:

1. Agreement on an initial view stems directly from the definition of $\mathsf{LocalView}_p(start_p)$, for every $p \in \mathsf{S}$.

2. The removal of a crashed process q from the view of every active process stems from the fact that q can have learned of only a finite number of actions before it crashed. If some p performs an infinite number of actions, there must be a point in its execution after which q was unresponsive to every announcement p made. Thus, q is not in $\mathsf{acks\text{-}rcvd}_p(\beta)$ and is then removed from $\mathsf{LocalView}_p(does_p(\beta))$.

3. By Lemma 4.2, if t processes are removed from $\mathsf{LocalView}_p$ at least one of them must have crashed (at most $t-1$ mistakes can have been made). We note that, since eventually all the processes in the system remove the crashed process from their view, these removals are uniform. ∎

Theorem 4.4 *WMS($t-1$) is sufficient for solving D-Uniformity with up to $f = N-t$ failures.*

Proof: Let MS be a WMS($t-1$) membership service and, in Figure 1, substitute it for line 7 (waiting for a majority of replies). This solution to D-Uniformity is f-resilient. Liveness is maintained because a live process p succeeds in communicating with every other live process within a finite time. Since MS reports every failure within a finite time, p does not wait for acknowledgments from crashed processes forever. Therefore, p performs within finite time every action it initiates.

The solution maintains safety because an action β, performed by process p, is acknowledged by at least one process that remains alive. Therefore, the information about β will eventually propagate to all the live processes. ∎

If removals are stable, a WMS($t-1$) membership service can make $t-1$ mistakes per process, or $N \times (t-1)$ mistakes globally. In [7] it is shown that for $t > 1$, such a failure detector is not strong enough to solve consensus when $f \geq \lceil N/2 \rceil$.[8]

When removals are not permanent, WMS($t-1$) can make infinitely may mistakes both about and to any process. Obviously, this membership service also cannot help in solving consensus.

4.3 $N-1$-Resiliency

For the special case of $f = N-1$, D-Uniformity is equivalent to Consensus.

Lemma 4.5 *The consensus problem is reducible to $N-1$-resilient D-Uniformity.*

Proof: A solution to $N-1$-resilient D-Uniformity requires a WMS(0). By definition any WMS(0) is a "perfect failure suspector" of [7], and consensus can be solved using the perfect failure suspector. ∎

By definition a WMS(0) makes no mistakes, and since all crashed processes are eventually removed, process removal with a WMS(0) is uniform.

5 Allowing Joins

In this section, we revert back to the asynchronous environment $\mathcal{A}$ without any membership service. As in Section 3, we examine the natural resiliency of D-Uniformity, but now consider the effect of adding processes to an execution in progress. Processes may be added to an ongoing execution for many reasons: to replace failed processes, to perform operations initiated by the application, and so forth. In this section, we show that process additions need not delay previously initiated D-Uniformity actions. We also show that process additions can be done locally in any order. In such an environment, D-Uniformity has solutions provided no more than a minority of the processes that have joined since the beginning are failed.

In an asynchronous environment, it is convenient to adopt the view that the process that joins was merely executing "slowly" for a period of time, during which it did not have permission to initiate system actions. *Joining* arises when such a process, say q, initiates communication with another process, say p, to obtain permission to participate in the system. Process q will be viewed as having *started* at that point in its protocol where it becomes possible for q to initiate new dynamically uniform actions, and to participate in the dynamic uniformity protocol for actions initiated by other system process. That is, the *start* event for q will now be preceded by protocol messages between q and p for the purpose of joining the system.

[8] In their terminology, for $t > 1$, WMS($t-1$) may make as many mistakes as the *strongly k-mistaken* failure detector, with $k = N \times (t-1)$.

The resulting model is illustrated in Figure 3. Here, we see process q issue a join request to process p, which is assumed to already be a member of the system. Process p responds by transferring state to q, reflecting any previously performed uniform actions, and then adds q to the system after running an appropriate protocol with other members (see below). We model this addition via a special event denoted $add_p(q)$. Of course, how a particular protocol chooses to transfer-state is up to the protocol designer, but we will assume that the state reflects any dynamically uniform actions known to p, so that q can also be viewed as having performed these actions. Process q, learning that it has been added, starts to participate as a full-fledged system member. This approach has the merit of closely modeling the behavior of real distributed systems (many of which support such a state transfer function), and is also natural (the requirement that q obtain permission is obviously necessary if partitions are to be avoided).

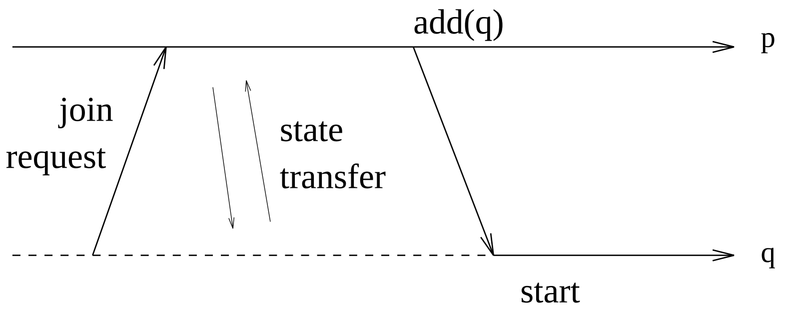

Figure 3: Adding of process q by p

Before we can capture the dynamic nature of the system, we must revise the definition given in Section 2 for the initial consistent cut, c_0. Above, we implicitly described a causal relationship between a new process and its state-transfer sponsor; our new definition will reflect this. Let S_0 be the subset of S of all processes with causally-independent *start* events:

$$\mathsf{S}_0 = \left\{ p \mid \nexists q \in \mathsf{S} : start_q \rightarrow start_p \right\} \subseteq \mathsf{S}.$$

Now take c_0 to be the *start* events of all processes in S_0. Certainly, this definition is consistent with the original definition and the assumption that all processes begin executing simultaneously.

We now introduce the *system view*, an indexical set defined as follows:

1. $\mathsf{SystView}(c_0) = \mathsf{S}_0$ and
2. $\mathsf{SystView}(c)$ is all processes that have been added to the local view of a process already in $\mathsf{SystView}(c')$ for some $c' < c$.[9]

$$\mathsf{SystView}(c) = \begin{cases} \mathsf{S}_0 & \text{if } c = c_0 \\ \left\{ q \mid \exists c' < c, p \in \mathsf{SystView}(c') : add_p(q) \in c \right\} & \text{for } c_0 < c \end{cases}$$

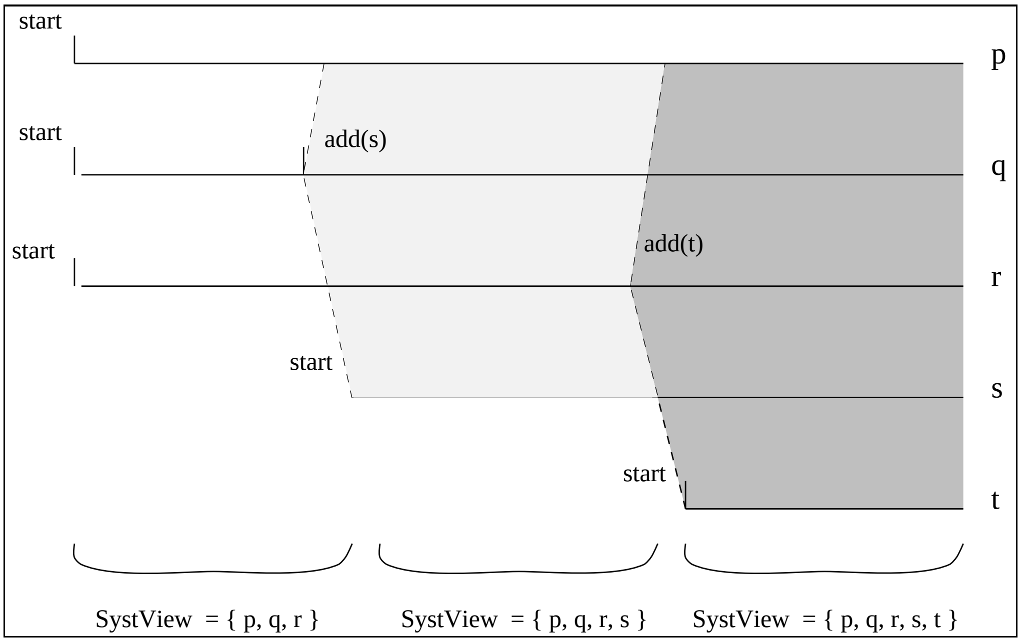

Figure 4: Development of $\mathsf{SystView}$

$\mathsf{SystView}()$ by this definition is monotonically increasing. Figure 4 depicts an example of the development of $\mathsf{SystView}$ in the system.

Only processes in $\mathsf{SystView}$ are allowed to perform actions, and Safety for D-Uniformity now becomes:

$$p \in \mathsf{SystView} \wedge \mathrm{DID}_p(\beta) \;\Rightarrow\; \bigwedge_{q \in \mathsf{S}} \Diamond\Big(\mathrm{DID}_q(\beta) \vee \mathrm{DISABLED}_q\Big) \;.$$

When an action β is performed, the D-Uniformity protocol must ensure that information about β persists in the current system view; any process joining later will receive information about β through the state-transfer operation. Note that, if any process in the system adds process q to the system before handling β, then q is part of the current system view, and should learn about β. Otherwise, all the processes have started handling β before adding q, and they could transfer the required information about β to q upon joining.

The liveness condition for D-Uniformity in the dynamic environment requires, in addition to liveness of uniform actions, that each process attempting to join the system is eventually added.

Clearly, a system that starts up with only a subset of S cannot hope to tolerate arbitrary crashes of a pre-defined, fixed number of processes; resiliency of the system should also be redefined to incorporate system size at various points in the execution. Another way to express this difference is to say that we limit the power of the adversary to crash processes within the context of the current $\mathsf{SystView}$, and do not take into account processes pending to enter the system. For simplicity, we assume that processes that attempt to join the system do not crash during the state-transfer phase, and thus the adversary can crash them only

[9] Given two consistent cuts c and c', $c' < c$ if and only if for each process mentioned in both c and c', its history component in c' is a prefix of its history component in c.

when they are added to the system[10]. To this end, we define the set $\mathsf{NotCrashed}(c)$ to be the subset of $\mathsf{SystView}(c)$ that have not crashed. We define the *liveness threshold* at c in terms of $\mathsf{NotCrashed}(c)$.

Definition 4 A solution $\mathcal{U}$ to D-Uniformity has a *l-threshold* if every cut c of every run of $\mathcal{U}$ is live provided $|\mathsf{NotCrashed}(c)| \geq l$.

For l-threshold where $l > \lfloor N/2 \rfloor$, the solution in Figure 1 works without change, regardless of what is being done with process additions. The solution is guaranteed to be live whenever more than $\lfloor N/2 \rfloor$ processes are alive (but cannot make progress until at least $\lfloor N/2 \rfloor + 1$ have started and joined the system).

For l-threshold where $l <= \lfloor N/2 \rfloor$, an argument similar to that of Lemma 3.3 shows that the problem is not solvable. However, in a dynamic environment, it makes sense to discuss a *varying* threshold degree, that increases along with additions to $\mathsf{SystView}$. The following lemma shows that *half*-threshold can be achieved:

Theorem 5.1 *If every cut c maintains* $|\mathsf{NotCrashed}(c)| > \lfloor |\mathsf{SystView}(c)|/2 \rfloor$ *then D-Uniformity is solvable.*

Proof: The algorithm in Figure 5 solves D-Uniformity in these conditions: There, a message is sent out containing the entire causal history of messages preceding it.

The following two lemmas contain the detailed proof of correctness of the protocol. Intuitively, Lemma 5.2 shows that there is a sequence of system views that maintain a quorum-intersection property, such that each view is established by a majority of the previous view. This "linearization" of the development of $\mathsf{SystView}$ occurs despite the fact that $\mathsf{SystView}$ may increase asynchronously, *i.e.*, despite the fact that a majority of the system may add generations of processes to the view, whereas processes in the minority remain oblivious to an arbitrary number of changes.

Using this form of linearization, Lemma 5.4 shows that information about any performed action reaches a majority of the $\mathsf{SystView}$, despite the fact that $\mathsf{SystView}$ may change concurrently to the action initiation and permission-gathering procedure.

The proof uses the following definitions:

- Define the *rank* of a process p as follows: If $p \in \mathsf{S}_0$, rank_p is zero. Else, let r be the maximal rank of processes q such that $\textsc{before}(start_q, start_p)$ holds; then rank_p is $r+1$.
- Denote S_0 by RV_0 ($RankView_0$), and RV_i to be the union of RV_{i-1} with all the processes of rank i.

Lemma 5.2 *Let s be of rank i, $i > 0$, and*

$$\mathsf{Predecessors}(s) = \left\{ q \in \mathsf{RV}_{i-1} \mid \mathrm{announce}(\mathrm{add}(q)) \rightarrow \mathrm{add}(s) \right\} \cup \mathsf{S}_0.$$

Then $\mathsf{Predecessors}(s)$ *contains* RV_{i-1}*, and a majority of* RV_{i-1} *has acknowledged* $\mathrm{add}(s)$ *before the event* $\mathrm{add}(s)$ *takes place.*

[10]This assumption could be relaxed, by assuming that a *majority* of the processes attempting to join the system at any point remain live. This would further complicate the discussion below.

Proof: Proof by induction. For the base case $i = 1$, $\mathsf{Predecessors}(s)$ contains RV_0 by definition. Also, processes of rank 1 are added by processes with rank 0. Therefore, in line 15 of the protocol, they wait for a majority of $\mathsf{LocalView}$, which is S_0.

Assume that the result holds for $i-1$ and continue for i. Let s be a process of rank i. Obviously, when $add(s)$ occurs, $\mathsf{FutureView}$ of the process performing $add(s)$ contains exactly those processes in $\mathsf{Predecessors}(s)$. Note that since rank_s is i, there exists at least one process of rank $i-1$, denoted q', $q' \in \mathsf{Predecessors}(s)$, such that $start_{q'} \rightarrow add(s)$ (which is stronger than $announce(add(q')) \rightarrow add(s)$).

By the induction hypothesis $\mathsf{Predecessors}(q')$ contains RV_{i-2} and therefore $\mathsf{Predecessors}(s)$ contains RV_{i-2}. Assume that there exists $r \in \mathsf{RV}_{i-1} \setminus \mathsf{RV}_{i-2}$, $r \notin \mathsf{Predecessors}(s)$. We show that this assumption leads to a contradiction.

By the induction hypothesis, adding r required a majority of acknowledgments from RV_{i-2}. Therefore, the additions of r and of q' "cross" at some process in RV_{i-2}. Formally, there exists a process $c \in \mathsf{RV}_{i-2}$, such that c acknowledges both $add(r)$ and $add(q')$ before the additions are performed. If c acknowledges $add(r)$ before $add(q')$, then $add(q')$ causally follows this acknowledgment, and therefore so does $add(s)$. Thus, $r \in \mathsf{Predecessors}(s)$, in contradiction to the assumption. Otherwise, c acknowledges $add(q')$ before $add(r)$. Thus, when r is added, q' is in $\mathsf{FutureView}$, and q' must have acknowledged r's addition. Since r and q' are of the same rank, this means that q' has acknowledged $add(r)$ before q' itself was added, *i.e.*, before $start_q$. Therefore, since $add(s)$ causally follows $start_{q'}$, we have $r \in \mathsf{Predecessors}(s)$. Once again, we reach a contradiction to the assumption.

Hence, there does not exist any $r \in \mathsf{RV}_{i-1} \setminus \mathsf{RV}_{i-2}$, such that $r \notin \mathsf{Predecessors}(s)$. As $\mathsf{Predecessors}(s)$ contains RV_{i-2}, $\mathsf{Predecessors}(s)$ contains RV_{i-1}. Also, when $add(s)$ is performed, all of RV_{i-1} is included in $\mathsf{FutureView}$, and thus, in line 15-16 of the protocol, at least a majority of acknowledgments are collected from RV_{i-1} before the addition. ∎

Corollary 5.3 *Let p be a process of rank i. Then by the time p starts executing (start_p), p has learned about every other i-ranked process.*

Proof: Let q be another i ranked process. Moreover, let process p be added by p' of rank $i-1$, and let process q be added by q' of rank $i-1$. Since a majority of RV_{i-1} is required for adding p and q, these majorities intersect. If q is announced at the intersection-process before p, then obviously p learns about q before it is added. If p is announced at the intersection-process before q, then p is in $\mathsf{FutureView}$ of q'. In this case q' can add q only after having received an acknowledgment from p: by definition this acknowledgment precedes the start of p (otherwise q would be of rank $i+1$). Again, p learns about q before it starts. ∎

Lemma 5.4 *Let p perform an action in event $\mathrm{does}_p(\beta)$. Let c be any consistent cut on which $\mathrm{does}_p(\beta)$ occurs (*i.e.*, $\mathrm{does}_p(\beta)$ is the last event of p's history component in c). Then a majority of $\mathsf{SystView}(c)$ knows β.*

Proof: Extending the notation, let

$$\mathsf{Predecessors}(\beta) = \Big\{q \mid announce(add(q)) \rightarrow does_p(\beta)\Big\} \cup \mathsf{S}_0.$$

Then when $does_p(\beta)$ occurs, $\mathsf{LocalView}_p$ contains exactly those processes in $\mathsf{Predecessors}(s)$. Since processes do not acknowledge regular actions before they start, $does_p(\beta)$ can only occur once the processes in $\mathsf{Predecessors}(s)$ have started.

Assume that i is the maximal rank such that $\mathsf{RV}_i \subseteq \mathsf{Predecessors}(\beta)$. Note that such i exists, since certainly $\mathsf{RV}_0 \equiv \mathsf{S}_0 \subseteq \mathsf{Predecessors}(\beta)$. By Corollary 5.3, there are no processes of rank higher than i in $\mathsf{Predecessors}(\beta)$. Thus, $\mathsf{RV}_i = \mathsf{Predecessors}(\beta)$, and a majority of RV_i is included in $\mathsf{acks\text{-}rcvd}_p(\beta)$.

Let q be a process of rank $i+1$, such that $q \in \mathsf{SystView}(c)$. By Lemma 5.2, a majority of RV_i has acknowledged q's addition to the system. Thus, the acknowledgments for β and for $add(q)$ "cross" at some process in RV_i. More precisely, there exists a process $c \in \mathsf{RV}_i$, such that $c \in \mathsf{acks\text{-}rcvd}_p(\beta)$ and c also acknowledges $add(q)$ before $add(q)$ is performed. If c acknowledges $add(q)$ before β, then $q \in \mathsf{Predecessors}(\beta)$, in contradiction to the above. Else, c acknowledges β before $add(q)$, and therefore in our model, q learns about β through the state-transfer operation. ∎

Proof of Theorem 5.1: From Lemma 5.4, it follows immediately that the protocol in Figure 5 maintains the D-Uniformity safety condition.

To see that the protocol is live, note that whenever a live process awaits responses from a majority of $\mathsf{LocalView}$ and from $\mathsf{FutureView}$, $\mathsf{FutureView} \supseteq \mathsf{SystView}$. Therefore, if less than $\mathsf{SystView}$ crash, it will eventually either receive responses from all of $\mathsf{FutureView}$ (before they start) or receive responses from a majority of them (after they start, and become members in $\mathsf{LocalView}$). ∎

Figure 5 demonstrates that it is possible to solve the D-Uniformity problem with joining if at each stage, a majority of the system persists. Moreover, this solution does not set a total order on process additions, nor guarantees the *virtually synchronous* execution model, in which process additions are totally ordered with respect to regular actions in the system (see [3] for a precise definition of the virtual synchrony model). For example, Figure 6 depicts a "disordered" scenario, in which processes p and r perform a process addition (of t) and a regular action (β) in different orders.

In the scenario in Figure 6, the system consists of four processes, p, q, r, s. p announces an action β, and waits for responses. Process q responds first, and then process r. Process r, having received p's announcement, q's response, and its own response (a majority), proceeds to perform β immediately. Meanwhile, r's acknowledgment reaches p only after another message is received from q, this message announcing the addition of a new process t. Process p inserts t into $\mathsf{FutureView}_p$, and will therefore delay execution of β until after t is added to the system.

In the practical systems we study, the membership protocols (*e.g.*, [10, 18]) require ordering of process additions, and enforce the virtual-synchrony property. Our study has demonstrated that these properties are not required for majority-threshold of uniform activity, and raises the question whether more efficient membership protocols can be developed to support this type of problems.

Finally, we note that addressing the issue of decreasing the liveness threshold below majority, by utilizing a powerful membership service, leads to a similar analysis to that of Section 4, and eventually to similar requisites on the number of mistakes allowed. Since the

```
        /* process p performs (each send piggybacks the entire causal history) : */

LocalView_p := S_0 ; FutureView_p := S_0 ;
if p \notin S_0 send announce(add(p)) messages and wait until received an add(p) event ;
    acknowledge and maintain add() messages while waiting ;

|| endless loop:
    select \beta fairly out of the following:
        1. \beta is a new action ;
        2. \beta is removed out from pending-buffer ;
    send(announce(\beta)) to all processes in LocalView_p ;          /* Initiate \beta */

|| upon receive(announce(\beta)) from q:
    send(ack(\beta)) to q ;
    if \beta is received for the first time
        if \beta is of the form add(r) add r to FutureView_p ;
        store \beta in pending-buffer ;

|| upon receive(ack(\beta)) from q:
    if collected ack(\beta) from \lfloor|LocalView_p|/2\rfloor + 1 processes out of LocalView_p and
    from all the processes in FutureView_p \ LocalView_p
        if \beta is of the form add(r)
            add r to LocalView_p and send add(r) to r ;              /* add_p(r) */
        else                                                     /* \beta is a regular action */
            perform \beta                                            /* does_p(\beta) */
        remove \beta from pending-buffer ;
```

Figure 5: A solution to D-Uniformity that has *half*-threshold and supports additions

technique and results are very similar, we omit them.

Remarks

All along, S has been a finite set of process identifiers. In theory then, system executions will inevitably terminate once the number of non-crashed processes falls below a majority. Making the name space countably infinite gives the potential for infinite (nonblocking) executions. However, the Safety clause becomes troublesome as it would now be quantifying over an infinite set. We can remedy this and remain faithful to the original interpretation of distributed coordination in the following manner. Let $c[p,\beta]$ be the 'latest' (with respect to the $<$ relation between cuts noted on page 12) consistent cut for which p's history component ends with the action $announce_p(\beta)$. Thus, $c[p,\beta]$ includes all the events whereby other processes receive p's announcement of its intention to perform β. Then define the *obligation set* of an action β being initiated by process p as

$$\mathsf{Obliged}(p,\beta) \quad = \quad \mathsf{SystView}(c[p,\beta])$$

and quantify over $\mathsf{Obliged}(p,\beta)$ in the Safety clause. In this way, $\mathsf{Obliged}(p,\beta)$ is always

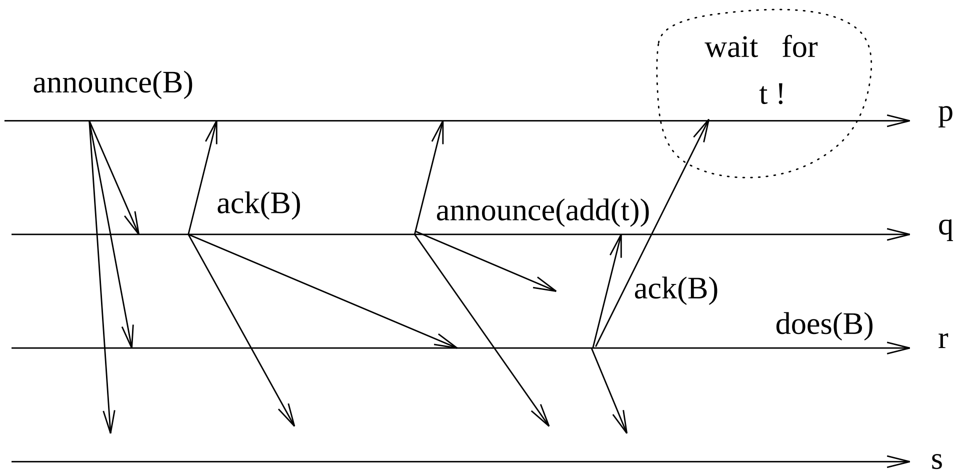

Figure 6: Disorder of regular actions and process additions

finite, is over time a piecemeal approximation to S, and is consistent with the algorithm given in Figure 5.

Finally, once an action is terminated at some cut c (*i.e.*, all obliged processes have either taken the action or are disabled at c), processes joining after c can only learn about the action by state-transfer. However, one can envision applications in which terminated actions are not important to the system (or are reflected in the process states) and so need not be propagated to late-joining processes. In these cases processes no longer need (unbounded) storage buffers for state-transfer, as the pending-actions buffers suffice. Phrasing Safety in terms of the indexical obligation set, rather than the entire process name space, allows us to consider these cases as well.

6 Conclusions

D-Uniformity is suitable for modeling the lowest layer in the distributed systems we have developed. The problem captures, in a very general way, the requirements of any system that will provide distinguished behavior in a primary partition, or guarantee that certain types of actions will eventually have system-wide effect. By studying the problem in an asynchronous environment composed of processes that join and leave while the system executes, substantial insight is gained into necessary and sufficient mechanisms for building fault-tolerant distributed software.

In our practical work, D-Uniformity arises in what is called the system membership layer, which is typically a component of the reliable multicast service. This layer is responsible for ensuring that the system can maintain its own liveness, by dynamically reconfiguring itself to exclude balky processes or processes that have been partitioned away from the system. The key requirement on such services is that they be able to distinguish the primary partition of a system, so that actions can be initiated on behalf of the system as a whole without risk that conflicting actions are being initiated in some other partition of the system. Given such a service, an elaborate programming model can be supported, including process groups, group communication, and group programming tools.

Our results show that systems with this structure can make progress provided that the number of mistakes made by their failure detection layer, as a function of the number of processes currently active in the system, is sufficiently small. Our work also sheds light on the protocol obligations for fault-tolerant distributed systems that operate in asynchronous environments. Finally, dynamic uniformity has implications for the implementation of uniform reliable multicast protocols, such as the protocols in [19] and the *safe* protocols of the Transis system.

Acknowledgments

We benefited from many discussions with Danny Dolev throughout this work, and accepted his valuable suggestions to the protocols and formulations in Section 5. We thank Fred Schneider for helpful comments on earlier versions of this paper.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] Rida Bazzi and Gil Neiger. The Possibility and the Complexity of Achieving Fault-Tolerant Coordination. In *Annual ACM Symp. on Principles of Distributed Computing*, number 11, pages 203–214, August 1992.

[3] K. P. Birman. *Reliable Distributed Computing with the Isis Toolkit*, chapter Virtual Synchrony Model. IEEE Press, 1994.

[4] K. P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138, Nov 87.

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

[6] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *proc. 11th annual ACM Symposium on Principles of Distributed Computing*, pages 147–158, 1992.

[7] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comp. Syst.*, 3(1):63–75, February 1985.

[9] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[10] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. submitted for publication. Available as CS TR94-6, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.

[11] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[12] Ajei Gopal. *Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination.* PhD thesis, Cornell University, 1992.

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

[14] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. TR 94-14, Inst. of Comp. Sci., The Hebrew University of Jerusalem, June 1994. submitted for publication.

[15] Gil Neiger and Mark Tuttle. Common Knowledge and Consistent Simultaneous Coordination. In *4th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LNCS, 486)*, number 4, pages 335–352, 1990.

[16] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of ACM*, 27(2):228–234, 1980.

[17] A. Ricciardi. *The Group Membership Problem in Asynchronous Systems.* PhD thesis, dept. of Computer Science, Cornell University, November 1992. (TR 92-1313).

[18] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[19] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, pages 561–568, May 1993.

[20] R. van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 27–28, April 1992.