Uniform Actions in Asynchronous Distributed Systems

Extended Abstract

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

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. It relates the problem to asynchronous Consensus, and shows that Consensus is a harder problem. We provide a rigorous characterization of the framework upon which several existing distributed programming environments are based. And, our work shows that progress is sometimes possible in a primary-partition model even when consensus is not.*

1 Introduction

Our work on fault-tolerant distributed systems (Isis [4], Transis [2], and Horus [18]) 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. In fact, 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 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. In D-Uniformity, there are certain actions that, if taken by any process in the system, must eventually be taken by every other active process; only failure and forcible removal (forced inactivity) excuse a process from taking the action.[1] Even actions taken by a process that later crashes or is forcibly removed must be propagated to the remaining active processes.

The motivation for 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 its that co-

*Institute of Computer Science, The Hebrew University of Jerusalem, Israel

†Computer Sci. Dept., Cornell University, Ithaca, NY, USA.
Supported by 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).

[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.

horts 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.

We then explore the impact 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 achieved with up to $N-t$ failures.

We then show that a membership service can allow dynamic *process-joins*. We show that in order to achieve resiliency with any single live member (*1-Threshold*), the membership service must ensure that process-joins are ordered with respect to regular actions. This property is indeed supported in practice by the virtual synchrony programming model, which is provided by the distributed systems we study.

In much prior work, *Distributed Consensus* is considered the basic problem in achieving distributed coordination [13]. For example, Chandra and Toueg show that Consensus is equivalent to both fault detection and to atomic broadcast [6]. It is well known that Consensus has no solution in asynchronous environments in which even a single process may crash [9]. As a result, Consensus *per se* is not the basic problem underlying all dynamic coordination in asynchronous environments. In particular, studying only Consensus ignores problems in which coordinating the activity of only a majority of the processes suffices. One of our goals is to better understand the relationship between ensuring progress in a *primary partition* and the classical Consensus problem.

Finally, to model the higher layers of abstraction found in the systems we study, we define two subclasses of Uniformity, for future research. The first consists of actions that must be done both uniformly and in the same *sequence*. This sequence problem has been studied extensively in the context of distributed database systems [1, 10, 11, 17]. The second subclass applies to uniform actions that must also eventually be known to have *terminated* (*i.e.*, taken by all processes that have not failed). This subclass is important in systems that perform some sort of clean-up action upon detecting termination (*e.g.*, re-using entries in a table, or discarding saved copies of messages related to the action).

The systems we have developed – Isis, Transis, and Horus – can be understood as practical tools for programming with dynamic uniformity. These systems can be viewed as having two layers. The upper layer assumes a failstop model [16] with notifications, upon which abstractions such as group communications and message delivery ordering are implemented. The lower layer implements a membership service that looks like a failstop failure detector to the upper layer, using agreement protocols to react to process join, completion, and 'suspected crash' events. The lower layer must track membership within a primary partition, an instance of D-Uniformity. By formalizing D-Uniformity, this paper demonstrates that the virtual synchrony programming model, shared by these three systems, rests on a sound foundation, and explains how virtual synchrony relates to the one within which asynchronous consensus has been studied.

2 System Model

The system consists of a finite set S of processes. 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 resend 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 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

[2] While similar in role to a *failure detector* [6], a membership service reports several types of membership events (*e.g.*, joins, and leaves) in addition to suspected failures.

[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.

for each $p \in \mathsf{S}$. The initial system cut, c_0, consists only of all the $start_p$ events. We assume familiarity with inter-event causality and the "happens-before" relation ($e \rightarrow e'$) [12], and with consistent cuts [7]. 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 [8], 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:

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

- $\textsc{did}_p(\beta)$ holds on c if and only if $does_p(\beta)$ is an event in p's history component of c,
- $\textsc{owns}_p(\beta)$ 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 $e_1 \rightarrow e_2$ 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):

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

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

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:

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

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

     /* Initiate β */
    send(announce(β)) to all the processes in S ;
    wait for ⌈N/2⌉ ack(β) s ;
    does_p(β);

|| 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.

Formula (1) does not require $\text{DID}_q(\beta)$ or 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.*$\bigvee_{p \in \mathsf{S}} \neg\text{DISABLED}_p$. An immediate consequence is that a process that is never disabled can eventually obtain permission to execute an infinite number of its actions.

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. ■

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 [9]. To see that D-Uniformity is solvable whenever consensus is solvable, assume $\text{CONSENSUS}(val)$ 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:

> choose a new action β;
> $res := \text{CONSENSUS}(\beta)$;
> 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 $c' \models \text{DID}_p(\beta)$, 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

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

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 [14]). 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].

[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

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 [5] 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 every execution of $\mathcal{U}$(MS) there is no consistent cut c such that*

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

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. ∎

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 $does_p(\beta)$.

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

Lemma 4.2 *If $\mathcal{U}$(MS) is an f-resilient solution to D-Uniformity, and* $\mathrm{does}_p(\beta)$ *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 $\{q_1, \ldots q_t\} \subseteq (S \setminus \mathsf{acks\text{-}rcvd}_p(\beta))$, 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. ∎

As in [6] 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 [6] 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 [6], 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

We now consider processes that join a computation. In an asynchronous environment, *late joining* arises when the start of a process, say q, was preceded by some communication with another process, say p. In this communication, presumably p has granted q permission to *join* the system, via a special event denoted $add_p(q)$. In this case, we deem it appropriate to relieve q from the obligation to perform actions that were initiated prior to $add_p(q)$. This approach is based on the following observations:

- A *state-transfer* operation, in which a newly joined process accepts a snapshot of the system upon joining, is a common practice in fault-tolerant distributed applications. State-transfer means that joining processes need not actually execute all relevant events to reach the desired local state.

- From a practical point of view, it is not feasible to expect processes to maintain information about actions indefinitely. Typically, when a process learns that all the currently active processes have performed some action, it discards that action from its buffers (Section 6 discusses this in more detail). Thus, existing processes need not store the accumulated execution history, on the chance that other processes may join a point far in the future.

[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 obligation set of an action β is the set of processes *obliged* to perform β. In order to define the obligation set we introduce a system view $\mathsf{SystView}$ defined as follows: (1) $\mathsf{SystView}$ contains initially the special set S_0 of processes that have their *start* event in the initial system cut c_0, and (2) $\mathsf{SystView}$ increases as new processes get the permission from processes in $\mathsf{SystView}$ to join. Formally:

Definition 4 Let c_0 be the initial system cut, and S_0 the special set of processes that have their start event in c_0. Let $add(q)$ be a special uniform action that gives q the permission to join. The system view $\mathsf{SystView}(c)$ on a cut c is recursively defined by:

1. $\mathsf{SystView}(c_0) \stackrel{\text{def}}{=} \mathsf{S}_0$;
2. if $p \in \mathsf{SystView}(c)$ and the event $add_p(q)$ is in a cut c, then p is in $\mathsf{SystView}(c)$.

We define now $\mathsf{Obliged}(\beta)$ as the set of processes that are in $\mathsf{SystView}$ before the initiation of action β propagates in the system:

Definition 5 Let $init(\beta)$ denote the initiation of action β by the owner of β; that is the event by which the owner of β announces β. The *obligation set* of β in run ρ is:

$$\mathsf{Obliged}(\beta, \rho) \stackrel{\text{def}}{=} \bigcup_{\{cut\ c \ \mid\ init(\beta) \notin c\}} \mathsf{SystView}(c)$$

Safety for D-Uniformity now becomes:

$$\mathrm{DID}_p(\beta) \Rightarrow \bigwedge_{q \in \mathsf{Obliged}(\beta)} \Diamond\Big(\mathrm{DID}_q(\beta) \vee \mathrm{DISABLED}_q\Big) .$$

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. To this end, we define the set $\mathsf{NotCrashed}(c)$ to be the subset of $\mathsf{SystView}(c)$ that have not crashed. We define resiliency at c is in terms of $\mathsf{NotCrashed}(c)$.

Definition 6 A solution $\mathcal{U}$ to D-Uniformity has a *t-threshold* if every cut c of every run of $\mathcal{U}$ is live provided $|\mathsf{NotCrashed}(c)| \geq t$.

5.1 One-Threshold Solutions

For simplicity, we show the weakest membership service required for a 1-Threshold (1-T) solution to D-Uniformity. The extension to the general case is done as in Section 4.

Lemma 5.1 *Let $p, q \in \mathsf{S}$, and let $\mathcal{U}(MS)$ be a 1-T solution to D-Uniformity. Then in no execution of $\mathcal{U}(MS)$ is there a consistent cut c such that*

$$c \models \text{DID}_p(\beta) \land q \in \mathsf{Obliged}(\beta) \land \neg\text{CRASH}_q \land \neg K_q\text{“}(\beta)\text{”}$$

Proof: Assume to the contrary that c satisfies the conditions above and occurs in some execution of $\mathcal{U}$. Immediately after c, suppose an adversary crashes all the processes in $\mathsf{SystView}(c)$ except q. This can be done if the protocol has a 1-Threshold, still leaving the system live. However, q may never learn about β (due to network assumptions); since $q \in \mathsf{Obliged}(\beta)$, this violates safety. ∎

Recall that $\mathsf{acks\text{-}rcvd}_p(\beta)$ is the set of processes from which p received some acknowledgment about β up to $does_p(\beta)$.

Lemma 5.2 *In any 1-Threshold solution to D-Uniformity, when p obtains permission for an action β, every process in $\mathsf{Obliged}(\beta) \setminus \mathsf{acks\text{-}rcvd}_p(\beta)$ has crashed.*

Proof: The proof is similar to that of Lemma 4.2. Let c be any cut on which $\text{DID}_p(\beta)$ holds. Assume there exists a process q that violates the Lemma. The adversary would choose to crash all processes except q on a cut c on which $\neg K_q$“(β)” holds, in contradiction to Lemma 5.1. ∎

Definition 7 An Open Weak Membership Service that supports joins, and makes zero mistakes (OWMS(0)), maintains the following properties:

1. Within a finite number of events after p starts, p obtains an initial membership view $\mathsf{LocalView}_p$, containing p and every process in S_0 (that has not crashed). In consequence, the initial local views of the processes of S_0 are identical (modulo failures).
2. Crashed processes are eventually removed from the local view of each active process.
3. No non-crashed process is removed from $\mathsf{LocalView}_p$.
4. Changes to $\mathsf{SystView}$ are ordered with respect to regular actions: For each process q, and each event $init(\beta)$, either an event $add(q)$ (by some process) causally precedes $init(\beta)$, or $init(\beta) \rightarrow add_r(q)$ for all r.

Theorem 5.3 *When processes joins are permitted, an OWMS(0) is weaker than any membership service extending the asynchronous environment $\mathcal{A}$ in which 1-threshold D-Uniformity has solutions.*

Proof: Consider the following, in which $\mathsf{LocalView}_p$ and changes to it are defined as follows:

Init Let β be the first action p gets permission to execute, and set p's initial local view to be $\mathsf{acks\text{-}rcvd}_p(\beta)$;

Add/Remove At each event $does_p(\beta)$, set $\mathsf{LocalView}_p(does_p(\beta))$ to $\mathsf{acks\text{-}rcvd}_p(\beta)$, but otherwise don't change $\mathsf{LocalView}_p$.

We now show that this membership service satisfies the properties of an OWMS(0):

1. If p is active, then an initial view exists for p as soon as p obtains permission for an action (say β). By Lemma 5.2, this view contains every process in $\mathsf{Obliged}(\beta)$ that has not crashed, and therefore includes every process in S_0 that has not crashed.
2. The proof of eventual removal of crashed processes is the same as in the proof of Theorem 4.3.
3. If a process q is in $\mathsf{LocalView}_p$ at some point, then in any action β initiated later by p, $q \in \mathsf{Obliged}(\beta)$. From Lemma 5.2, if q is removed from $\mathsf{LocalView}_p$, q must have crashed.
4. Assume to the contrary that in some run ρ, the event $init(\beta)$ occurs at p concurrently to the addition of a process q, $add_s(q)$, such that q is unknown to p. Immediately after $add_s(q)$ let the adversary crash s. Note that, in the run ρ, the process q belongs to $\mathsf{Obliged}(\beta)$.

   Let ρ' be an identical run except that in ρ' the process s crashes immediately **before** the event $add_s(q)$. In ρ', $q \notin \mathsf{Obliged}(\beta)$. We claim that ρ and ρ' are indistinguishable to the process p, and therefore, in one of them, $\mathsf{acks\text{-}rcvd}_p(\beta)$ cannot reflect $\mathsf{Obliged}(\beta)$ correctly, in contradiction to Lemma 5.2. ∎

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

/* *Initiate* β */
5 for each event $add(q)$ in the past
add q to $\mathsf{LocalView}_p$ (unless q was removed);
6 $\mathsf{LV}_p \leftarrow \mathsf{LocalView}_p$
7 send(announce(β)) to all processes in LV_p;
8 wait for ack(β) from each process
in $\mathsf{LV}_p \cap \mathsf{LocalView}_p$;
9 $does_p(\beta)$;

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

Figure 2: A D-Uniformity protocol using an OWMS(0) and supporting joins.

We note that, as every correct process initiates infinitely many actions, process additions are done uniformly by OWMS(0). This follows from item 4 of Definition 7.

Theorem 5.4 *An OWMS(0) is sufficient for building a 1-Threshold solution for D-Uniformity in the face of process joins.*

Proof: The protocol in Figure 2 uses an OWMS(0) to solve D-Uniformity, and has a 1-Threshold.

If p initiates β at line 7, then $\mathsf{LV}_p = \mathsf{Obliged}(\beta)$. The reason is that by property 4 of OWMS(0), any process addition has either preceded $init(\beta)$, or will causally follow it. In the former case, the process is included in LV_p, and in the latter case, it is precluded from $\mathsf{Obliged}(\beta)$. Thus, by line 9, every process in $\mathsf{Obliged}(\beta)$ that has not crashed knows about β, which guarantees uniformity.

Liveness is ensured by item 2 of the definition of OWMS(0) and by line 8: p will never wait indefinitely to get the permission to perform β. ∎

6 Sequence and Termination

D-Uniformity characterizes a problem in which the actions taken do not in any way conflict with each other; this assumption of no contention is not always reasonable. For example, in totally-ordered multicast services the delivery of a message conflicts with the delivery of all other messages in that only one of them can occupy the i^{th} space in the delivery order. Whenever actions β, γ conflict, D-Sequential-Uniformity adds the following safety condition to D-Uniformity:

$$\text{BEFORE}(does_p(\beta), does_p(\gamma)) \wedge \text{DID}_q(\gamma) \Rightarrow \text{BEFORE}(does_q(\beta), does_q(\gamma)) \ .$$

Even if p performs an action β, it can take arbitrarily long for the action to *terminate*; that is, to be performed by all active obliged members. Knowing eventually β will terminate (the case in any solution to D-Uniformity) differs from eventually knowing β has terminated, which requires an additional global state detection. Definitive knowledge of termination is a pragmatic concern and arises whenever information about actions is kept in a buffer. Eventually the buffer must be cleaned, and terminated actions are reasonable candidates to remove. D-Terminated-Uniformity adds the following safety condition:

$$\text{DID}_p(\beta) \Rightarrow \Diamond\text{DISABLED}_p \vee \bigwedge_{q \in \mathsf{Obliged}(\beta)} \Diamond\Big(K_p\text{DID}_q(\beta) \vee K_p\text{DISABLED}_q\Big)$$

Whereas D-Uniformity can sometimes be solved when Consensus cannot, D-Sequential-Uniformity and D-Terminated-Uniformity are equivalent to Consensus. This raises the question of exactly how a membership service relates to the various D-Uniformity problems. It seems most appropriate to define a family of membership services with varied D-Uniformity properties, which are then reflected in the higher level software built over the membership layer. The membership services used in the systems cited here are not identical, but all provide some combination of Uniformity, Sequence, and Termination properties.

Note that when a primary partition membership service is actually implemented (*e.g.*, in Isis), then, lacking access to a sufficiently accurate failure-detector, the service might sometimes block, meaning that it is impossible to make progress without potentially violating the safety conditions of D-Uniformity. Since the upper layers then block as well, there are failure scenarios in which these systems can be prevented from making progress.

7 Conclusions

D-Uniformity is suitable for modeling the lowest layer in the distributed systems we study. This layer implements a reliable-multicast service, which ensures uniform delivery of messages multicast within a group, and is used both to maintain membership information within a primary partition of the system, and in applications structured as process groups. Our results show that a reliable multicast can be implemented within any majority partition of the system, or, given access to a membership service, can be implemented with an even higher degree of resiliency. Most interestingly, given a membership layer that can make an infinite number of mistakes, but at most $t-1$ mistakes at each event, the reliable multicast layer can tolerate $N-t$ failures. In addition, a membership layer can support dynamic process additions. We show that in order to achieve 1-Threshold resiliency, this membership layer needs to maintain order between process joins and regular actions.

Our results have implications for the implementation of uniform reliable multicast protocols, such as the protocols in [15] and the *safe* protocols of the Transis system. Moreover, these results apply to the reliable multicast protocols employed within the "strong" membership services of Isis and Transis, which go well beyond the weak membership services employed in this paper.

Acknowledgments

We thank Fred Schneider and Danny Dolev for their many helpful discussions.

References

[1] A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *ACM SIGACT-SIGMOD Symp. on Prin. of Database Systems*, pages 240–251, March 1986.

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

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

[10] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Syst.*, 4(1):32–53, February 1986.

[11] M. Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *ACM Trans. Comp. Syst.*, 5(3):249–274, August 1987.

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

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

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

[15] 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.

[16] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comp. Syst.*, 2(2):145–154, May 1984.

[17] D. Skeen, F. Christian, and A. El Abbadi. An Efficient Fault-Tolerant Algorithm for Replicated Data Management. In *ACM SIGACT-SIGMOD symp. on prin. of Database Systems*, pages 215–229, March 1985.

[18] 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.