The Actions-Constraints approach to replication: Definitions and Proofs

Marc Shapiro, Karthik Bhargavan
Microsoft Research Cambridge
7 J J Thomson Ave., Cambridge CB3 0FB, UK
tel. +44 1223 479 739, fax +44 1223 479 999

27 March 2004

Technical Report
MSR-TR-2004-14

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract: *Replicated information raises the major issue of consistency. We have developped a simple, formal framework, in order to better understand and compare consistency properties of replication protocols. The framework is both formal and implementable. Our language is simple enough to prove interesting properties, yet sufficiently powerful to specify diverse systems.*

In our model, each site maintains its local view of data, of actions to execute, and of the constraints that define legal execution schedules. Adding actions increases the number of possible schedules; adding constraints reduces scheduling non-determinism. We exhibit significant subsets of actions that are progressively more determined and show a number of useful properties. The system is consistent if every action is eventually scheduled and local executions converge. We compare different possible formulations of the consistency property and prove them to be mutually equivalent. This underscores the deep commonalities between diverse protocols. One of our formulations can be used to characterise consistency in partially replicated systems, i.e., where a site has visibility of only a subset of data, actions and constraints. Finally, we show how a number of protocols from the literature are modeled in the action-constraint framework.

1 Introduction

In a distributed system, information is often replicated, but since a site's view of remote state is partial and possibly stale, consistency is a major issue. Despite the large variety of replication protocols [18], we lack a common framework for understanding and comparing them. It is especially difficult to reason about optimistic systems and about partial replication.

This paper presents such a framework, based on actions, i.e., reified operations, and constraints, i.e., reified invariants that the system is responsible for maintaining. Our constraints are very simple and have been implemented efficiently. Specification of a replicated system is modular in that users, applications, objects, and protocols each contribute some part of the constraints.

Within this framework, we identify the significant building blocks that are common to all replication protocols. We study several properties relevant to consistency in protocols. We give a precise meaning to the intuitive concepts underlying consistency, and to consistency itself. We compare four different formulations of consistency, including Eventual Consistency from the literature, and Mergeability, a generalisation of serialisability. We show that they are all mutually equivalent. We extend the Mergeability conditions to encompass partial replication. The Mergeability formulation clarifies that in the general case, consistency entails consensus; however we study sufficient conditions for consistency that can be evaluated locally, and describe a new protocol based on this insight. Furthermore, we model and we prove the consistency properties of diverse kinds of protocols from the literature: a timestamped-write protocol, serialisable transactions, and a partial replication transaction protocol.

At a deep level, all consistent replication algorithms are equivalent since they

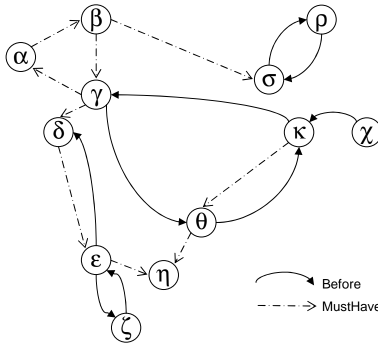

Figure 1: Example constraints. α, β and γ form a *parcel*, an atomic (i.e., all-or-nothing) execution. γ executes only if δ also executes. δ is *causally dependent* on ϵ. ϵ and ζ *mutually exclude* each other. Only two actions out of the three γ, θ and κ can execute. If both χ and κ execute, χ comes first.

are based on the same primitives and satisfy the same formal properties. They differ mainly by the subset of the constraint repertoire that they use and by the transition rules.

The paper proceeds as follows. Section 2 provides a high-level overview. Then we review the main data structures and their properties in Section 3. Then we move on to how a replicated system and consistency in Section 4. Partial replication is the topic of Section 5. Section 6 applies our approach to some protocols from the literature. Section 7 compares with related work, and Section 8 contains conclusions and suggestions for future work.

2 Overview

This section overviews of our approach informally (later sections define all the terms precisely). The main objects are actions, or reified operations, and constraints, which constrain scheduling. Each site maintains a local view of known actions and constraints, called the site's *multilog*.[1] A site's multilog evolves as local clients submit actions and constraints, and as it receives remote multilog contents. The current local state is the result of running a valid schedule computed from the current local multilog. Adding actions increases the number of possible schedules; adding constraints reduces the scheduling non-determinism.

Constraints are of two types: ordering ($\rightarrow$) and implication ($\triangleright$). One might imagine other useful constraint types; what we have is both sufficiently simple to prove useful properties and sufficiently powerful for our current purposes. Our choice of constraints enables the implementation of an efficient, optimising scheduler [15].[2] The same constraint language specifies invariants at different

[1] A log is an ordered record of local actions [9]. A multilog is unordered, contains actions from several sites, and also contains constraints.

[2] The present choice of constraints does not support dynamic data dependencies, whereby

levels: user intents, for instance "run action β only if α succeeds;" data semantics, for instance "action α and β mutually exclude each other" (conflict), and replication protocol decisions, such as "serialise action α before β."

Figure 1 shows some example constraints. The user has ensured that the three actions α, β and γ will either execute or non-execute atomically with cycle $\alpha \triangleright \beta \wedge \beta \triangleright \gamma \wedge \gamma \triangleright \alpha$. Action δ depends causally upon ϵ, shown by $\epsilon \rightarrow \delta \wedge \delta \triangleright \epsilon$. Actions ϵ and ζ conflict; the cycle $\epsilon \rightarrow \zeta \wedge \zeta \rightarrow \epsilon$ ensures that they mutually exclude each other. We have described practical applications and systems using constraints elsewehere [15, 20].

A replication protocol propagates actions and constraints to the different sites in a manner described by *transition rules* written in first-order logic. Furthermore, the protocol may decide to add constraints to ensure consistency. For instance, to ensure once and for all that action ϵ is executed, it would add $\text{INIT} \triangleright \epsilon$ to some local multilog. Before deciding, the protocol might negociate between sites (in this example, to choose between executing ϵ or ζ); our model does not directly deal with this negociation, modeled as temporary non-determinism and/or as protocol-specific transition rules.

Multilogs grow monotonically: once added, an action or constraint is never removed. Hence, only irrevocable decisions appear as constraints (this simplifies the logic, and anyhow, revocable information does not contribute to correctness).

3 The data structures and their properties

We define the different data structures used in the formalism and study their properties. This section makes no reference to sites or to time; these will be added later on.

3.1 Actions and schedules

An *action* is an instance of an arbitrary operation (which can read or write shared data) with its parameters. Once submitted, an action does not change. Actions are application-specific and not interpreted by our framework. Actions are assumed deterministic: executing the same action twice from the same state has the same result.

We note A the (finite) set of actions $\alpha, \beta, \gamma \dots$. We distinguish a special action INIT. To α corresponds the *non-action* noted $\overline{\alpha}$, a placeholder with no effect.

A *schedule* is an ordered set of actions and non-actions that starts with INIT:

Definition 1 (Schedule) *A schedule $S = (L, <_S)$ consists of a set of actions $L \cup \{\text{INIT}\}$, where $L \subseteq A$, a mapping from L to $\{0, 1\}$, and a strict total order $<_S$, where $\text{INIT} <_S x$ for all $x \in L$.*

---

an action can fail if the shared state violates some arbitrary predicate. We can express the fact that action α fails and cannot be scheduled, by setting $\alpha \rightarrow \alpha$, but the dynamic dependency is lost. We discuss alternatives in Section 8.

The intuition is the following. L is a set of actions available for scheduling. Depending on factors to be explained shortly, the system orders the actions $\alpha \in L$ and and chooses, for each one, whether to *execute* it (mapped to 1, action $\alpha \in S$) or to *non-execute* it (mapped to 0, non-action $\overline{\alpha} \in S$). Hence our notations:

$$\left.\begin{array}{r}\alpha \in S \\ S = \lambda\alpha\mu\end{array}\right\} \ \alpha \text{ executes in } S$$

$$\left.\begin{array}{r}\overline{\alpha} \in S \\ S = \lambda\overline{\alpha}\mu\end{array}\right\} \ \alpha \text{ non-executes in } S$$

$$\mathit{sched}(\alpha, S) \text{ iff } \alpha \in L \quad \alpha \text{ is } \textit{scheduled} \text{ in } S$$

where λ and μ are arbitrary sequences (possibly empty) of actions and non-actions.

The schedule containing only INIT is called the empty schedule. All other schedules are non-empty.

Let us note as $\alpha.op$ the operation corresponding to action α, and the initial state as ϵ. The state after executing $S = \text{INIT}\,\alpha\,\beta\,\gamma$ is $\mathit{State}(S) = \gamma.op(\beta.op(\alpha.op(\epsilon)))$.

Most actions commute with one another, i.e., the result of $\mathit{State}(S\alpha\beta) = \mathit{State}(S\beta\alpha)$ for any S. The non-commutativity property indicates this is not the case:

Definition 2 (Non-Commutativity) *The non-commutativity relation $\leftrightarrow$ is defined over $A \times A$. Two actions α and β do not commute iff the result of $\alpha\beta$ may differ from $\beta\alpha$:*

$$\begin{aligned}\alpha \leftrightarrow \beta \stackrel{\text{def}}{\equiv} \exists S \text{ such that} \\ (S\alpha\beta \neq \bot \vee S\beta\alpha \neq \bot) \\ \wedge\, S\alpha\beta \neq S\beta\alpha\end{aligned}$$

A non-action has no effect: $\forall\alpha \in A, \forall S : \mathit{State}(S\,\overline{\alpha}) = \mathit{State}(S)$. It commutes with every action or non-action.

Two schedules are *equivalent* if they execute and non-execute the same actions, and non-commuting actions execute in the same order in both.

Definition 3 (Schedule Equivalence) *Schedules S and T are* equivalent, *if and only if they contain the same actions and non-actions, and non-commuting actions execute in the same order:*

$$\begin{aligned}S \equiv T \iff & (\forall\alpha \in A : \alpha \in S \Leftrightarrow \alpha \in T \wedge \overline{\alpha} \in S \Leftrightarrow \overline{\alpha} \in T) \\ & \wedge(\forall\alpha, \beta \in A : \alpha \leftrightarrow \beta \wedge \alpha \in S \wedge \beta \in S \Rightarrow (\alpha <_S \beta \Rightarrow \alpha <_T \beta))\end{aligned}$$

This is an equivalence relation over schedules. This definition of equivalence is stronger than necessary but sufficient for our purposes.

3.2 Multilogs and constraints

A *multilog* is a data structure containing actions and constraints. A schedule is sound if it is safe with respect to the contents of some multilog. We define constraints formally herein as a relation between actions in a schedule.

A site's local view of the global state is a distinguished multilog, the site-multilog; it executes some site-schedule computed from the site-multilog.

Formally, a multilog is a data structure $M = (K, \rightarrow, \triangleright)$ where:

- $K \subseteq A$ is a set of actions,
- $\rightarrow$ (the Before constraint) is a relation over $A \times A$ (not acyclic, nor reflexive, nor transitive),
- $\triangleright$ (the MustHave constraint) is a transitive and reflexive relation over $(A \cup \{\text{INIT}\}) \times (A \cup \{\text{INIT}\})$.

K is called the set of *actions known* in M, and $\rightarrow$ and $\triangleright$ together are called the *constraints known* in M. The set of all multilogs is noted $\mathcal{M}$.

Both constraint types are relations between two actions and a schedule. $\rightarrow$ represents an ordering constraint on schedules: if $\alpha \rightarrow \beta$ then α may never follow β in any schedule. $\triangleright$ is an implication constraint: if $\alpha \triangleright \beta$, then executing α implies to execute also β.

Definition 4 (Well formed schedules) *Schedule $S = (L, <_S)$ is* well formed *with respect to multilog $M = (K, \rightarrow, \triangleright)$ iff all actions known in M are scheduled, and only those, i.e., if $L = K$.*

Definition 5 (Sound schedules and sound multilogs) *S is* $\rightarrow$-sound *with respect to M iff it is well formed and it obeys M's Before constraints: $\forall \alpha, \beta \in A : \alpha \rightarrow \beta \implies (\alpha \in S \wedge \beta \in S \Rightarrow \alpha <_S \beta)$.*

S is $\triangleright$-sound *if it is well formed and obeys M's MustHave constraints: $\forall \alpha, \beta \in A : \alpha \triangleright \beta \implies (\alpha \in S \Rightarrow \beta \in S)$.*

Schedule S is sound *with respect to M if it is both $\rightarrow$-sound and $\triangleright$-sound. The set of sound schedules with respect to M is noted $\Sigma(M)$.*

A multilog M is sound *iff it can generate a sound schedule, i.e., iff $\Sigma(M) \neq \emptyset$.*

For instance, the multilogs $(\{\alpha\}, \emptyset, \{\text{INIT} \triangleright \alpha\})$ and $(\{\alpha\}, \{\alpha \rightarrow \alpha\}, \emptyset)$ are both sound, whereas $(\{\alpha\}, \{\alpha \rightarrow \alpha\}, \{\text{INIT} \triangleright \alpha\})$ is not.

Lemma 1 *If S is a sound schedule: $\alpha \triangleright \beta \implies (\overline{\beta} \in S \Rightarrow \alpha \notin S)$.*

Proof: By contradiction: Assume $\overline{\beta} \in S$ and $\alpha \in S$. Then $\beta \in S$; but then S is not a schedule, a contradiction.

3.3 Extension and union of multilogs

A multilog $M' = (K', \rightarrow', \rhd')$ *extends* $M = (K, \rightarrow, \rhd)$, noted $M \subseteq M'$, if all known actions and constraints of M are in M': $(K \subseteq K') \wedge (\rightarrow \subseteq \rightarrow') \wedge (\rhd \subseteq \rhd')$. The extends relation is a partial order over $\mathcal{M}$.

We define a union operation over multilogs.

Definition 6 (Multilog union) *Let $M_1 = (K_1, \rightarrow_1, \rhd_1)$, $M_2 = (K_2, \rightarrow_2, \rhd_2)$ and $M = (K, \rightarrow, \rhd)$. The union operation is defined by:*

$$M = M_1 \cup M_2 : \begin{cases} K = K_1 \cup K_2 \\ \rightarrow\, = (\rightarrow_1 \cup \rightarrow_2) \\ \rhd = (\rhd_1 \cup \rhd_2) \end{cases}$$

The notation $M \cup K'$ (where $K' \subseteq A$) is used as shorthand for $M \cup (K', \emptyset, \emptyset)$. Similarly, $M \cup \{\alpha \rightarrow \beta\}$ is shorthand for $M \cup (\emptyset, \{(\alpha, \beta)\}, \emptyset)$, and $M \cup \{\alpha \rhd \beta\}$ is shorthand for $M \cup (\emptyset, \emptyset, \{(\alpha, \beta)\})$.

For any multilogs M_1 and M_2, it is the case that $M_1 \subseteq M_1 \cup M_2$. Note that the union of sound multilogs is not necessarily sound.[3]

3.4 Equivalence of multilogs; congruence

Definition 7 (Equivalent Multilogs) *Two multilogs are equivalent if they generate the same set of sound schedules. $M_1 \equiv M_2$ iff $\Sigma(M_1) = \Sigma(M_2)$*

Lemma 2 (Multilog Equivalence and $\rhd, \rightarrow$) *Suppose $M = (K, \rightarrow, \rhd)$. Then the following are true:*

1. $M \equiv M \cup \alpha \rhd \alpha$
2. *If $\alpha \rhd \beta$ and $\beta \rhd \gamma$, then $M \equiv M \cup \alpha \rhd \gamma$.*
3. *If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ and* INIT $\rhd \ldots \rhd \beta$, *then $M \equiv M \cup \alpha \rightarrow \gamma$.*

The proof is left to the reader.

Although Before is not transitive in general, intermediate guaranteed actions make it transitive: If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ and $\beta \in \mathit{Guar}(M)$ then $\alpha \rightarrow \gamma$.

Definition 8 (Congruence) *A equivalence relation $\mathcal{R} \subseteq M \times M$ on multilogs is said to be a congruence if: whenever $M \mathcal{R} M'$ then for all extensions E, $(M \cup E) \mathcal{R} (M' \cup E)$.*

Lemma 3 *Multilog equivalence $\equiv$ is a congruence.*

This means that we can replace a multilog by an equivalent multilog at any time because no present or future decision can be affected. To simplify upcoming proofs we identify M and its equivalence class.

[3] Remember we are identifying a multilog with its equivalence class. So $M \subseteq M'$ really means that there exists $M'' \equiv M'$ such that $(K \subseteq K'') \wedge (\rightarrow \subseteq \rightarrow'') \wedge (\rhd \subseteq \rhd'')$.

3.5 Significant subsets

In general, when an action is first submitted, it is not known whether it will execute nor in what order, as this will be decided later during the execution of the replication protocol. For instance actions can be aborted and reordered [7, 22]. This uncertainty is represented in our framework as non-deterministic scheduling. Eventually however the status of each action becomes more determined as constraints are added. Note that constraints are irrevocable.[4]

The *significant subsets* in Definition 9 contain actions whose scheduling is partially or completely determined, and constitute the basic building blocks of any replication protocol.

Definition 9 *The significant subsets of M are the following:*

Guaranteed: *$Guar(M)$ is the smallest set satisfying: (1) $\text{INIT} \in Guar(M)$. (2) $\forall \beta \in A$: if $\alpha \in Guar(M)$ and $\alpha \rhd \beta$, then $\beta \in Guar(M)$.*

Dead: *$Dead(M)$ is the smallest set satisfying: (1) $\forall \alpha \in A$: if $\beta_1, \ldots, \beta_m \in Guar(M)$, where m is any natural integer, and $\alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha$, then $\alpha \in Dead(M)$. (2) $\forall \alpha \in A$: if $\beta \in Dead(M)$ and $\alpha \rhd \beta$, then $\alpha \in Dead(M)$.*

Serialised: $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in A | \forall \beta \in A, \alpha \nleftrightarrow \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M)\}$

Decided: $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$

Stable: $Stable(M) \stackrel{\text{def}}{=} Dead(M) \cup \{\alpha \in Guar(M) \cap Serialised(M) | \forall \beta \in A : \beta \rightarrow \alpha \Rightarrow \beta \in Stable(M)\}$

An action becomes *guaranteed* or *dead* when it is irrevocably executed or non-executed, respectively. It becomes *serialised* when irrevocably ordered with respect to non-commuting, non-dead actions. These two transitions can occur in any order.

Once it is either dead, or both guaranteed and serialised, it is said *decided*: both its execution or non-execution status and its ordering relative to other actions are final. An action is *stable* when either dead, or guaranteed and all its predecessors are themselves stable. (We shall prove, in Section 4.3, that decided actions eventually become stable.) Stability means that the outcome of the action will never change. In a practical implementation, stable actions can be garbage-collected from multilogs.

3.5.1 Guaranteed and dead actions

We consider a multilog $M = (K, \rightarrow, \rhd)$ and the associated sound schedules $\Sigma(M)$, and some multilog M' that has the same constraints. An action is

[4]This simplifies the logic, and besides, revocable information would not contribute to correctness.

guaranteed iff it executes in every sound schedule of any M', and dead iff it executes in no sound schedule of any M'.

Theorem 1 *Consider a multilog $M = (K, \rightarrow, \rhd)$. If M is sound:*

- $Guar(M) = \{\alpha \in A \cup \{\text{INIT}\} | \forall K' : K' \subseteq A, \forall S \in \Sigma(M') : \alpha \in S\}$, *where* $M' = (K', \rightarrow, \rhd)$.
- $Dead(M) = \{\alpha \in A | \forall K' \subseteq A, \forall S \in \Sigma(M') : \alpha \notin S\}$, *where* $M' = (K', \rightarrow, \rhd)$.

Note how these subsets are characterised by the known constraints, and are independent of known actions. Proof of the first equality is left as an exercise for the reader. The second proof follows.

Dead⇒not executed: Assume $\beta_1, \ldots, \beta_m \in Guar(M)$ and $\alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha$. Then any schedule S where $\alpha \in S$ violates $\rightarrow$-soundness. If $\gamma \rhd \beta$ where $\beta \in Dead(M)$ then γ violates $\rhd$-soundness. Thus, neither α nor γ may execute in a sound schedule.

Dead⇐not executed We wish to prove that, if α is not executed in any sound schedule, then $\alpha \in Dead(M)$. By contradiction: assume $\alpha \notin Dead(M)$. Also assume $\forall K' \subseteq A, \forall S \in \Sigma(M') : \alpha \notin S$, where $M' = (K', \rightarrow, \rhd)$. For $\alpha \notin K'$ nothing can be concluded. However, for $\alpha \in K'$ the above is equivalent to $\forall S \in \Sigma(M') : \overline{\alpha} \in S$. Choose one such K' and one such S. We can write $S = \lambda\overline{\alpha}\mu$. Since $\alpha \notin Dead(M)$ then for every cycle $\alpha \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha$, there exists i such that $\beta_i \notin Guar(M)$. If β_i is in λ, let λ' be the same as λ with β_i replaced with $\overline{\beta_i}$, and equal to λ otherwise. Similarly for μ. Let $S' = \lambda'\alpha\mu'$. Since S is $\rightarrow$-sound, S' is also. Furthermore, since $\alpha \notin Dead(M)$ and S is $\rhd$-sound, S' is also $\rhd$-sound. Conclusion: S' is sound, a contradiction.

QED

If the multilog is sound, a guaranteed action must be known: $Guar(M) \subseteq K$.

To make some action α become guaranteed, it is sufficient to add the constraint $\text{INIT} \rhd \alpha$.

Lemma 4 (Guaranteed and Dead sets are disjoint) *M is a sound multilog if and only if no action is both guaranteed and dead:* $\Sigma(M) \neq \emptyset \iff Guar(M) \cap Dead(M) = \emptyset$.

Proof:

(⇒) By contradiction. Consider schedule $S \in \Sigma(M)$. Consider action $\alpha \in Guar(M) \cap Dead(M)$. Since $\alpha \in Guar(M)$, then $\alpha \in S$. Since $\alpha \in Dead(M)$, either $\alpha \rightarrow \beta_0 \rightarrow \ldots \rightarrow \beta_m \rightarrow \alpha$ (where the β_i are guaranteed), or $\exists \beta \in Dead(M) : \alpha \rhd \beta$ such that $\beta \rightarrow \beta$. In the either case, S is not sound, a contradiction.

($\Leftarrow$) Assume $Guar(M) \cap Dead(M) = \emptyset$. Hence $\forall \alpha \in A : \text{INIT} \rhd \alpha \Rightarrow \neg(\alpha \rightarrow \alpha) \wedge \forall \beta \in Dead(M) \neg(\alpha \rhd \beta)$. Construct a schedule S that executes only guaranteed actions: $\alpha \in K \setminus Guar(M) \Rightarrow \overline{\alpha} \in S$. Order S according to Before: this is possible since we just assumed there are no cycles of $\rightarrow$ and that $\rhd$ can be satisfied. Furthermore, S is closed for $\rhd$. Hence S is sound, and $\Sigma(M) \neq \emptyset$.

QED

Lemma 5 ($\rightarrow$ must be acyclic in $Guar(M)$) *M is a sound multilog if and only if there is no cycle of $\rightarrow$ where all actions are guaranteed: $\Sigma(M) \neq \emptyset \iff \nexists \alpha_1, \ldots, \alpha_n, n > 0 : \alpha_1, \ldots, \alpha_n \in Guar(M) \wedge \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_1$.*

The proof is immediate, from Lemma 4.

It follows that $\alpha \rightarrow \alpha \Rightarrow \alpha \in Dead(M)$. To make an action α become dead, it is sufficient to add the constraint $\alpha \rightarrow \alpha$.

Lemma 6 ($\rightarrow$ Acyclic implies Sound) *If the relation $\rightarrow$ is acyclic, $Dead(M) = \emptyset$. Then M is sound.*

Proof: The property $Dead(M) = \emptyset$ derives from the definition of the dead set. If $Dead(M) = \emptyset$ then $Guar(M) \cap Dead(M) = \emptyset$, or (according to Lemma 4) M is sound.

It may tempting to conclude that if all actions commute, this implies soundness. However this is not true, because even commuting actions may be ordered by $\rightarrow$. Here is a counter-example: suppose α and β commute, they are both guaranteed, and $\alpha \rightarrow \beta \wedge \beta \rightarrow \alpha$.

Another tempting conjecture is that a multilog is sound if all cycles of $\rightarrow$ contain a dead action. This is untrue because, if the multilog is unsound, a dead action might also be guaranteed, as in the following counter-example: $(\{\alpha\}, \{\alpha \rightarrow \alpha\}, \{\text{INIT} \rhd \alpha\})$.

Theorem 2 *Consider some multilog $M \in \mathcal{M}$. For any multilog $M' \in \mathcal{M}$ that extends it ($M \subseteq M'$) the following properties are true:*

- $Guar(M) \subseteq Guar(M')$
- $Dead(M) \subseteq Dead(M')$
- $Serialised(M) \subseteq Serialised(M')$
- $Decided(M) \subseteq Decided(M')$
- $Stable(M) \subseteq Stable(M')$
- $\Sigma(M) = \emptyset \Rightarrow \Sigma(M') = \emptyset$

If follows that if M is not sound, M' remains unsound.

The proof is left as an exercise to the reader.

3.5.2 Serialised and decided actions

An action is *serialised before* another if the former always executes before the latter (in any schedule where they both execute). Serialisation is defined only between non-commuting actions: if $\alpha \leftrightarrow \beta$, action α is serialised before β if and only if $\alpha \rightarrow \beta$ (remember that we identify a multilog with its equivalence class).

Note that the "serialised before" relation is not transitive nor anti-symmetric; indeed it is possible to have both $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$.

A non-dead action α is *serialised in M* if it is ordered relative to all non-commuting actions that execute. Once the status of an action cannot change, i.e., once it is guaranteed/dead and (if guaranteed) serialised, it is said *decided.*

3.6 Prefixes

We study several kinds of prefixes. A simple prefix schedule conforms to the intuitive definition. A strong prefix is a prefix of all sound schedules of a multilog. A monotonic strong prefix remains a strong prefix of all extensions of its defining multilog.

P is a prefix of sound schedule $S \in \Sigma(M)$, noted $P \ll S$, iff common non-commuting actions are in the same order, and additional actions of S come after those of P.

Formally:

Definition 10 (Prefix of a schedule) *Consider some multilog $M \in \mathcal{M}$, a schedule $S = (L, <_S)$ where $S \in \Sigma(M)$, and another schedule $P = (J, <_P)$.*

$$
\begin{aligned}
P \ll S \stackrel{\text{def}}{\equiv} \forall \alpha, \beta \in A :\\
&\alpha \in P \Rightarrow \alpha \in S\\
&\wedge \alpha \in P \wedge \beta \in P \wedge \alpha \leftrightarrow \beta \wedge \alpha <_P \beta \Rightarrow \alpha <_S \beta\\
&\wedge \alpha \in P \wedge \beta \notin P \wedge \beta \in S \wedge \alpha \leftrightarrow \beta \Rightarrow \alpha <_S \beta
\end{aligned}
$$

If P is a prefix of S, one can write $S = P\lambda$.

Lemma 7 (Prefix and equivalence) *Consider prefix P of schedule S: Then any schedule P' equivalent to P is also a prefix of S, and P is a prefix of any schedule S' equivalent to S.*

$$P \ll S \implies (P' \equiv P \Rightarrow P' \ll S) \wedge (S' \equiv S \Rightarrow P \ll S')$$

The proof is left as an exercise for the reader, by direct application of the definition of prefix and equivalence.

3.6.1 Strong prefix of a multilog

An interesting special case is when P is a prefix of *all* sound schedules in $\Sigma(M)$; we say P is a *strong prefix* of M, which we note $P \ll \Sigma(M)$.

Definition 11 (Strong Prefix)

$$P \ll \Sigma(M) \stackrel{\text{def}}{\equiv} S \in \Sigma(M) \Rightarrow P \ll S$$

The empty schedule (the schedule containing only INIT) is a strong prefix of any sound multilog.

Lemma 8 (Strong prefix, guaranteed and dead) *Consider schedule P that is a strong prefix of multilog M: $P \ll \Sigma(M)$.*

1. *An action executed in P is guaranteed in M: $\forall \alpha \in A : \alpha \in P \Rightarrow \alpha \in Guar(M)$*
2. *An action non-executed in P is dead in M: $\forall \alpha \in A : \overline{\alpha} \in P \Rightarrow \alpha \in Dead(M)$*

Proof: 1: $\alpha \in P \Rightarrow (\forall S \in \Sigma(M), \alpha \in S) \Leftrightarrow \alpha \in Guar(M)$. 2: $\overline{\alpha} \in P \Rightarrow (\forall S \in \Sigma(M), \overline{\alpha} \in S) \Leftrightarrow \alpha \in Dead(M)$. **QED.**

Note that actions in a strong prefix are not necessarily decided. Here is a counter-example: consider $A = \{\alpha, \beta\}$ such that $\alpha \not\leftrightarrow \beta$, and $M = (\{\alpha\}, \emptyset, \{\text{INIT} \rhd \alpha\})$. Then $P = \text{INIT}\,\alpha$ is a strong prefix of M even though α is not decided.

Lemma 9 (Strong Prefix and Before) *Consider action α executing in $P \ll \Sigma(M)$. Consider an action β such that $\beta \rightarrow \alpha$. Then either β executes in P, or β is dead. Formally: $\alpha \in P \wedge \beta \rightarrow \alpha \implies \beta \in P \vee \beta \in Dead(M)$*

Proof: By contradiction. Consider α and β such that $\alpha \in P \wedge \beta \rightarrow \alpha \wedge \beta \notin P \wedge \beta \notin Dead(M)$. Since β is not dead, there exists a schedule $S \in \Sigma(M) : \beta \in S$. Since $\beta \rightarrow \alpha$ it follows that $\beta <_S \alpha$. Then P is not a strong prefix of M, a contradiction. **QED.**

Lemma 10 (Prefix and serialisation order) *The order of execution in a strong prefix $P \ll \Sigma(M)$ of non-commuting actions is their serialisation order:*

$$\forall \alpha, \beta \in A : \alpha \not\leftrightarrow \beta \wedge \alpha, \beta \in P \implies (\alpha <_P \beta \Rightarrow \alpha \rightarrow \beta)$$

Proof: If $\alpha \rightarrow \beta$ then $\alpha <_P \beta$. To prove the converse (by contradiction), assume $\alpha <_P \beta$ and $\alpha \not\leftrightarrow \beta$ and $\neg(\alpha \rightarrow \beta)$. Consider some sound schedule S; if $P \ll \Sigma(M)$ then $S = \lambda\alpha\mu\beta\nu$. The schedule $S' = \lambda\beta\mu\alpha\nu$ is sound but P is not a prefix of S', a contradiction.

Contrary to intuition, it is not the case that the set of decided actions form a strong prefix. Here is a counter-example: $\alpha \rightarrow \beta \rightarrow \gamma$ with $\{\alpha, \gamma\} \subseteq Guar(M)$ and $\beta \notin Guar(M) \cap Dead(M)$. It is always possible for an action that is known but undecided, and commutes with all decided actions, to be scheduled arbitrarily within the decided actions. In contrast, as we shall see shortly, stable actions form a strong prefix.

3.6.2 Monotonic Strong Prefix

We are particularly interested in systems where a strong prefix remains a strong prefix when the multilog is extended. We give a sufficient condition for extension that maintains this property. The we show the relation between the monotonicity property, and stability.

Definition 12 (Monotonic extension of multilog) *Consider a multilog M. Another multilog M' is a* monotonic extension *of M if strong prefixes of M remain strong prefixes of M': $M \subseteq M' \wedge M \ll \Sigma(P) \Rightarrow M' \ll \Sigma(P)$.*

Not all extensions are monotonic in general, but the following weaker lemma is always true.

Lemma 11 (Partial prefix monotonicity) *Assume $P \ll \Sigma(M)$ and $M \subseteq M'$. The the following properties are true for all $S' \in \Sigma(M')$.*

1. $\alpha \in P \Rightarrow \alpha \in S'$
2. $\overline{\alpha} \in P \Rightarrow \overline{\alpha} \in S'$
3. $\alpha \in P \wedge \beta \in P \wedge \alpha \not\leftrightarrow \beta \wedge \alpha <_P \beta \Rightarrow \alpha <_{S'} \beta$

Proof: 1: $\alpha \in P \Rightarrow \alpha \in \mathit{Guar}(M) \Rightarrow \alpha \in \mathit{Guar}(M') \Rightarrow \alpha \in S'$. 2: similar reasoning. 3: $\alpha \in P \wedge \beta \in P \wedge \alpha \not\leftrightarrow \beta \wedge \alpha <_P \beta \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow' \beta \Rightarrow \alpha <_{S'} \beta$. **QED**.

The following theorem provides a sufficient condition for an extension of a multilog to be a monotonic extension.

Theorem 3 (Monotonicity of strong prefix w.r.t. extension) *Consider a multilog $M = (K, \rightarrow, \rhd)$, a strong prefix $P \ll \Sigma(M)$, and a multilog $M' = (K', \rightarrow', \rhd')$ that extends M : $M \subseteq M'$. If every new action (i.e., every action in K' not already scheduled in P) either is dead or is serialised after P, then P remains a strong prefix of M'. Formally, $P \ll \Sigma(M')$ if for all $\alpha, \beta \in A$: $(\alpha \in P \wedge \beta \in K' \wedge \neg \mathit{sched}(\beta, P)) \Rightarrow \alpha \rightarrow' \beta \vee \beta \in \mathit{Dead}(M')$.*

The theorem follows directly from Lemma 11.

Theorem 4 (A monotonic strong prefix is stable) *Let $M \in \mathcal{M}$ be a multilog, and schedule P be a strong prefix of M: $P \ll \Sigma(M)$. If all multilogs M' that extend M are monotonic extensions of M, then every action scheduled in P is stable:*

$$\forall \alpha \in A : \mathit{sched}(\alpha, P) \implies \alpha \in \mathit{Stable}(M)$$

Proof: First we use induction over the length of P. Let's write $P = x_0 x_1 \dots x_n$ where $x_0 = \textsc{init}$ and either $x_j = \alpha_j$ or $x_j = \overline{\alpha}_j$ for all j between 1 and n.

- Induction hypothesis: $\alpha_0 \in \mathit{Stable}(M)$. Trivially true.

- Induction condition on m:

$$(\forall k, 0 \leq k < m \leq n, \alpha_k \in \mathit{Stable}(M)) \implies \alpha_m \in \mathit{Stable}(M))$$

  Assume $x_m = \overline{\alpha}_m$; then α_m is dead (Lemma 8), hence trivially stable. Alternatively, assume $x_m = \alpha_m$; then α_m is guaranteed. Furthermore, any α_k such that $k < m$ is before: $\alpha_k <_P \alpha_m$, and that is their serialisation order if they do not commute (Lemma 10). Conversely, any $\alpha_{k'}$ such that $m < k'$ is guaranteed but serialised afterwards; it follows that $\neg(\alpha_{k'} \rightarrow \alpha_m)$.

By the same reasoning, any action α in P is serialised before any non-commuting action β such that $\beta \notin P$ but $\beta \in S' \in \Sigma(M')$ unless $\beta \in \mathit{Dead}(M')$. **QED**.

Theorem 5 (Stable actions form a strong monotonic prefix) *Consider a sound multilog M and a sound schedule $S \in \Sigma(M)$. Construct schedule P such that $\alpha \in P \Leftrightarrow \alpha \in S \wedge \alpha \in \mathit{Stable}(M)$ and $\overline{\alpha} \in P \Leftrightarrow \overline{\alpha} \in S \wedge \alpha \in \mathit{Stable}(M)$ and $\alpha, \beta \in P \wedge \alpha <_M \beta \Rightarrow \alpha <_P \beta$. Then $P \ll \Sigma(M)$, and, for all M' such that $M \subseteq M'$, $P \ll \Sigma(M')$.*

Proof: Consider some $S \in \Sigma(M')$. Any action executed in S but not in P must be after P in S. (1) $\alpha \in P \Rightarrow \alpha \in \mathit{Guar}(M) \Rightarrow \alpha \in S$. (2) The execution order in P is the serialisation order, therefore $\alpha <_P \beta \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha <_M \beta$. (3) Consider non-commuting actions $\alpha \leftrightarrow \beta$ such that $\alpha \in P$ and $\beta \in S \wedge \beta \notin P$. Since $\alpha \in \mathit{Stable}(M)$ it is serialised with respect to β. The case $\alpha \rightarrow \beta$ is not possible since $\beta \notin \mathit{Stable}(M)$, therefore $\beta \rightarrow \alpha$. This is true for all S and all M' such that $M \subseteq M'$. Furthermore, $\mathit{Stable}(M) \subseteq \mathit{Stable}(M')$. **QED**.

4 Evolution and consistency of distributed state

A replicated system is a collection of sites, with one distinguished multilog per site, called site-multilogs. Each site computes sound schedules from its local multilog, called site-schedules. Clients update their local site-multilog by submitting actions and constraints. Sites exchange messages to update their multilog contents. This evolution of site-multilogs over time is subject to transition rules. The universal transition rules presented in this section simply describe client submissions and messaging and apply to all replicated systems. Specific protocols may have their own additional transition rules.

4.1 Site-multilog and site-schedule

Each site i has a distinguished *site-multilog* that evolves over time t, noted $M_i(t) = (K_i, \rightarrow_i, \rhd_i)(t)$. The state of site i is the result of executing some site-schedule $S_i(t) \in \Sigma(M_i(t))$. Since there may be many sound schedules for a given site-multilog, the current state is non-deterministic.

As the site-multilog evolves over time, the site-schedule also evolves to remain sound.[5]

We study the properties of $M_i(t)$ over time, as the client at site i submits new actions and/or constraints, as the replication protocol adds constraints, and as sites communicate with one another. For simplicity we use a global time notation but at no loss of generality, because we do not assume that a site can observe the global time. Also with no loss of generality we assume discrete time.

For simplicity we use a global time notation but we do not assume that a site can observe the global time. With no loss of generality we assume discrete time. The following *transition rule* describes submissions and messaging and applies to every protocol.

4.2 Universal transition rule

We describe replication protocols as rules governing how site-multilogs may change over time. The following rule applies to every protocol: site-multilogs change either because a local client submits new actions or constraints, or by receiving contents from a remote site-multilog.

The site-multilog $M_i(t)$ takes new values over time only according to the following legal transitions, where i represents an arbitrary site, t an arbitrary time, and $\alpha, \beta, \gamma \ldots$ arbitrary actions:

Transition Rule 1 (Universal transition rule) *Site-multilog $M_i(t)$ evolves over time according to the following legal transitions:*

1. Nothing happens: $M_i(t+1) = M_i(t)$
2. A local client submits new actions or constraints: $M_i(t+1) = M_i(t) \cup (\{\alpha_1, \ldots, \alpha_l\}, \{(\beta_1, \gamma_1), \ldots, (\beta_m, \gamma_m)\}, \{(\delta_1, \epsilon_1), \ldots, (\delta_n, \epsilon_n)\})$, *where* $l \geq 0, m \geq 0, n \geq 0$.
3. The site receives contents from a remote site-multilog: $M_i(t+1) = M_i(t) \cup (\{\alpha_1, \ldots, \alpha_l\}, \{(\beta_1, \gamma_1), \ldots, (\beta_m, \gamma_m)\}, \{(\delta_1, \epsilon_1), \ldots, (\delta_n, \epsilon_n)\})$, *where* $l \geq 0, m \geq 0, n \geq 0$ *and where* $\exists j \neq i, t' \leq t$ *such that* $\alpha_1, \ldots, \alpha_l \in K_j(t')$ *and* $\{(\beta_1, \gamma_1), \ldots, (\beta_m, \gamma_m)\} \subseteq \rightarrow_j (t')$ *and* $\{(\delta_1, \epsilon_1), \ldots, (\delta_n, \epsilon_n)\}) \subseteq \rhd_j(t')$.

By design, the three transitions are not always distinguishable. For instance, when K_i adds an action already known in K_j, it is indifferent whether the clients at i and j submitted the same action redundantly, or whether it was submitted at j and transmitted from j to i. The separate transitions are defined here as they will be useful in the definition of eventual consistency (Section 4.4.1).

Later (Definition 13) we will introduce a property called mergeability, implying that every transition from $M_i(t)$ to $M_i(t')$, where $t' > t$, preserves soundness.

Lemma 12 (Site multilog monotonically extends) *A site multilog monotonically extends itself over time, i.e.,* $t < t' \implies M_i(t) \subseteq M_i(t')$.

[5] We do not assume $S_i(t) \ll S_i(t+1)$). If this "prefix property" [22] is desired, it can be obtained using appropriate constraints. Otherwise, the implementation should be capable of rollback and replay. Once an action is stable, it will not be rolled back.

The proof is immediate by the transition rules.

Theorem 6 (Guaranteed, dead, serialised, decided are monotonic)

$$\begin{aligned}\forall i \in \mathit{sites}, t < t' \implies & \mathit{Guar}(M_i(t)) \subseteq \mathit{Guar}(M_i(t')) \\ & \wedge \mathit{Dead}(M_i(t)) \subseteq \mathit{Dead}(M_i(t')) \\ & \wedge \mathit{Serialised}(M_i(t)) \subseteq \mathit{Serialised}(M_i(t')) \\ & \wedge \mathit{Decided}(M_i(t)) \subseteq \mathit{Decided}(M_i(t')) \\ & \wedge \Sigma(M_i(t)) = \emptyset \Rightarrow \Sigma(M_i(t')) = \emptyset\end{aligned}$$

The proof is immediate by Lemma 12 and Theorem 2.

A strong prefix does not in general remain a strong prefix over time. However, according to Theorem 3, if all new actions are serialised after the current strong prefix, then it remains a strong prefix over time. According to Theorem 5, the stable actions form a strong prefix that remains over time.

4.3 Liveness conditions

A system satisfies Eventual Action Propagation (EAP) if the actions $K_i(t)$ known at i at time t are eventually known at arbitrary site j: $\exists t' : K_i(t) \subseteq K_j(t')$. Eventual Constraint Propagation (ECP) is similar.

We specify the liveness conditions that will be useful when discussing consistency.

Eventual Action Propagation is the property that any action known at some site eventually becomes known at every other site.

Property 1 (Eventual Action Propagation (EAP)) *Consider the set of known action sets $K_i(t)$, varying over sites and time. For any pair of sites i, j and for any time t, there exists a time t' when $K_i(t) \subseteq K_j(t')$.*

The similar property for constraints follows.

Property 2 (Eventual Constraint Propagation (ECP)) *Consider the set of known constraint sets $\rightarrow_i (t)$ and $\rhd_i(t)$, varying over sites and time. For any pair of sites i, j and for any time t, there exists a time t' when $\rightarrow_i (t) \subseteq \rightarrow_j (t') \wedge \rhd_i(t) \subseteq MustHave_j(t')$.*

The above properties are insufficient to guarantee significant progress. Hence the more relevant property:

Property 3 (Eventual Decision) *A replicated system has the Eventual Decision property, if every submitted action is eventually decided: $\alpha \in K_i(t) \Rightarrow \exists t' : \alpha \in \mathit{Decided}(M_i(t'))$*

Note that the Eventual Decision property is local to site i.

Eventual Decision does not preclude the trivial implementation that makes every action dead. This would be difficult to rule out formally, since an action may always fail (see Footnote 2).

Theorem 7 (Eventual decision implies eventual stability) *If a system has the eventual decision property, then any known action is eventually stable:*

$$\begin{array}{l}(\forall \alpha \in A, \alpha \in K_i(t) \Rightarrow \exists t' : \alpha \in \mathit{Decided}(K_i(t'))) \\ \quad \Longrightarrow \\ (\forall \alpha \in A, \alpha \in K_i(t) \Rightarrow \exists t'' : \alpha \in \mathit{Stable}(K_i(t'')))\end{array}$$

Proof: Consider some $\alpha \in \mathit{Decided}(K_i(t))$. α is serialised with respect to all β such that $\alpha \nleftrightarrow \beta$. Consider a chain of guaranteed actions $\ldots \beta_i \rightarrow \ldots \beta_j \ldots \rightarrow \alpha$. By soundness, we know it to be acyclic, hence it has a minimal element β_0. By assumption, eventually β_0 is decided; since β_0 is minimal it has no predecessor by $\rightarrow$; therefore when it is decided it is also stable. If β_0 is the only predecessor of β_1, then β_1 is stable, and so on by induction. If β_1 has several predecessors, proceed by induction over the number of predecessors and by induction over their own predecessors. **QED**.

4.4 Consistency

In this section, we define some interesting properties on replicated systems: the Eventual Consistency, the Common Monotonic Strong Prefix property, and Mergeability. We show that Eventual Consistency and the Common Monotonic Strong Prefix property are equivalent. Both imply Eventual Action Propagation. Taken togethter with Eventual Action Propagation, the properties of Eventual Decision and Mergeability are also equivalent to the Common Monotonic Strong Prefix property, and hence to Eventual Consistency. Thus we have four equivalent, alternative defintions of the intuitive notion of consistency.

4.4.1 Eventual consistency

We now compare different formulations of the consistency property. The first is Eventual Consistency, which has been used in the literature to argue informally the correctness of optimistic replication systems [2]. A system is Eventually Consistent if, if every client stops submitting actions (and, presumably, constraints), then eventually every site will reach the same final value.

The first part says that there is some time T after which Transition 1.2 of the universal transition rule (Section 4.2) does not fire. The second part says that by some time T', the site-schedules at every site are equivalent, and remain equivalent at all later times. Formally:

Property 4 (Eventual consistency) *A system is Eventually Consistent if, if every client stops submitting, and submitted actions are decided, then eventually every site will reach the same sound final value:*

$$\begin{array}{l}\exists T : \forall i, t > T \Rightarrow \textit{Transition 1.2 does not fire at } i \\ \quad \Longrightarrow \\ \exists T', \forall t', t'', i, j : t' > T' \wedge t'' > T' \wedge S_i(t') \in \Sigma(M_i(t')) \wedge S_j(t'') \in \Sigma(M_j(t'')) \\ \qquad\qquad \Rightarrow S_i(t') \equiv S_j(t'')\end{array}$$

There are two objections to this definition. One, in real systems, clients are not expected to ever stop submitting. Two, it provides no indication to implementors how to achieve the consistency goal.

Note that an additional liveness condition is necessary to make Eventual Consistency a useful property.

4.4.2 Common Monotonic Strong Prefix (CMSP)

We propose a different formulation of consistency, seeking to avoid the drawbacks just noted.

Property 5 *A replicated system $M_i(t)$ (i varying over sites, t over time) satisfies the CMSP Property if there exists a function $\pi(i,t)$ such that:*

1. *π is a strong prefix: $\pi(i,t) \ll \Sigma(M_i(t))$*
2. *The CMSP is equivalent at all sites: $\forall i,j,t : \pi(i,t) \equiv \pi(j,t)$*
3. *The CMSP is monotonically non-shrinking over time: $\forall i,t,t' : t < t' \implies \pi(i,t) \ll \pi(i,t')$*
4. *Every known action eventually reaches the prefix: $\forall \alpha \in A, t, i : \alpha \in K_i(t) \implies \exists t' : sched(\alpha, \pi(i,t'))$*

$\pi(i,t)$ is called Common Monotonic Strong Prefix (CMSP) at site i and time t.

The CMSP property implies Eventual Action Propagation. We now prove that the CMSP Property is equivalent to the conjunction of the Eventual Consistency and Eventual Decision properties.

Theorem 8 *If every client stops submitting, then a replicated system that satisfies the Eventual Consistency property if and only if it satisfies the CMSP property.*

EC $\Rightarrow$ CMSP: Assume Eventual Consistency (EC); if clients stop submitting, by some time T, the final state at site i is $S_i(T) \in \Sigma(M_i(t))$. By EC, for all i, j, $t', t'' > T$, $S_i(t') \equiv S_j(t'')$. The following function satisfies CMSP: $\pi(i,t) = \begin{cases} \text{INIT} & \text{for } t < T \\ S_i(T) & \text{for } t \geq T \end{cases}$

CMSP $\Rightarrow$ EC: Assume the CMSP Property. Every action is eventually stable, hence Eventual Decision is ensured. Suppose that the client at site i stops submitting at time t_i. By time $T_0 = \max_i(t_i)$ all clients have stopped submitting. Let $A' = \bigcup_j K_j(T_0)$. According to the CMSP Property, there is a time where every action in A' is in the prefix: $\forall \alpha \in A', \exists t'_i : sched(\alpha, \pi(i, t'_i))$. Assume sound site-multilogs; there exists sound site-schedules. Since no new actions are submitted, A' does not change, and the prefix covers the whole site-schedule: $S_i(t'_i) \equiv \pi(i, t'_i)$. According CMSP, all prefixes are equivalent; by time $T' = \max_k t'_k$ the prefix covers the whole site-schedule at every site; hence the site-schedules are mutually equivalent: $\forall i,j : t' > T', t'' > T' \Rightarrow S_i(t') \equiv \pi(i,t') \equiv \pi(j,t'') \equiv S_j(t'')$. **QED**.

4.4.3 Mergeability

It is heartening that the two properties above are equivalent, but implementors would still like something more operational. Hence the mergeability property, which states that all combinations of site-multilogs remain sound.

Definition 13 (Mergeability) *A system has the Mergeability property if, given any arbitrary collection of sites* $i, i', i'' \ldots$ *and any arbitrary collection of times* $t, t', t'' \ldots : M_i(t) \cup M_{i'}(t') \cup M_{i''}(t'') \ldots$ *is sound.*

Mergeability means that sites may not (even at different times) take mutually unsound decisions. This is easy to ensure in a centralised system, but not in a distributed one. For instance, consider at time t, Site 1 has multilog $(\{\alpha\}, \emptyset, \{\text{INIT} \triangleright \alpha\})$ and Site 2 has multilog $(\{\alpha\}, \{\alpha \to \alpha\}, \emptyset)$. They are both sound but not mergeable, as their union $(\{\alpha\}, \{\alpha \to \alpha\}, \{\text{INIT} \triangleright \alpha\})$ is not sound.

More generally, if $\alpha \in \mathit{Guar}(M_i(t))$ for some i and t, then it cannot be the case without violating mergeability that $\alpha \in \mathit{Dead}(M_{i'}(t'))$ for any i and t'. Similarly, if it is the case that both $\alpha \to_i (t)\beta$ and $\beta \to_{i'} (t')\alpha$, then it cannot also be the case that both $\alpha \in \mathit{Guar}(M_i(t))$ and $\beta \in \mathit{Guar}(M_{i'}(t'))$. Thus, the decision to make an action guaranteed or dead, or what order to serialise two guaranteed actions, or to guarantee a serialised action, entails a global agreement.

Assuming liveness properties Eventual Decision and Eventual Action Propagation, then Mergeability (a safety property) us equivalent to the Eventual Consistency property:

Theorem 9 *If every client stops submitting, then a replicated system satisfies Eventual Consistency if and only if it satisfies the Mergeability, Eventual Decision and Eventual Propagation properties.*

Proof: Consider a system that has the Mergeability property. Assume clients stop submitting actions at time t; at this time, the multilogs have contents $M_i(t) = (K_i, \to_i, \triangleright_i)(t)$ for i varying over the set of sites. By Eventual Decision and Theorem 7, there exists a time t_i where all actions are stable at i. By Eventual Action Propagation, there exists a time T where, for all j in the set of sites, $M_j(T) = \bigcup_i M_i(t_i)$. By mergeability, this union is sound. Since the actions are stable, the final state is uniquely defined. Conversely, assume Eventual Consistency. Make clients stop submitting at some arbitrary set of times t_i. There is a time T where all sites have converged to a uniquely defined common state, i.e., site schedules $S_i(T)$ are stable and mutually equivalent. Then every action is decided and the $M_i(T)$ are sound and mutually equivalent, which implies Eventual Action Propagation. **QED**.

Theorem 12 shows that mergeability is a generalisation of serialisability.

4.4.4 Uniform Local Soundness

With a slightly stronger liveness condition, Mergeability reduces to the Uniform Local Soundness property, i.e., that site-multilogs are sound at all times.

Property 6 *A replicated system has the Uniform Local Soundness property iff multilogs are sound at all times:* $\forall i, t : \Sigma(M_i(t)) \neq \emptyset$.

Theorem 10 *Assuming the Eventual Decision and Eventual Action Propagation properties, a system satisfies Uniform Local Soundness and Eventual Constraint Propagation if and only if it satisfies Mergeability.*

Proof: (ULS+ECP $\Rightarrow$ Mergeability) Assume Eventual Action and Constraint Propagation, and Eventual Decision. Assume the system satisfies ULA but not Mergeability. Then there exists $i, i', i'', \ldots$ and $t, t', t'', \ldots$ such that $M_i(t) \cup M_{i'}(t') \cup M_{i''}(t'') \ldots$ is unsound. By Eventual Action Propagation and Eventual Constraint Propagation, there exists a time T where (say) $M_i(T) \subseteq M_i(t) \cup M_{i'}(t') \cup M_{i''}(t'') \ldots$. By Theorem 2, $M_i(T)$ is unsound, a contradiction. (Mergeability $\Rightarrow$ ULS+ECP) Assume Eventual Action Propagation and Eventual Decision. We already know that Mergeability implies eventual consistency, which itself implies ULS. Since the local schedules are equivalent, this means the multilogs are equivalent. By the definition of multilog equivalence, ECP is implied. **QED**.

4.5 Deciding

The intuitive meaning of Mergeability is that the system must decide for each action, without sites making conflicting decisions. In the general case then, decisions to make dead, guarantee or serialise entails a distributed consensus, a known difficult problem [8]. In practice the most straightforward way to avoid conflicts is to centralise all decisions at a single primary site [22, 18] (the primary may change over time).

However, an important design goal for a distributed system is autonomy of sites. There are well-known protocols that manage to make progress without an online consensus algorithm. For instance the Timestamp Ordering protocol of Section 6.1 uses commutativity, and the ESDS protocol (Section 6.4) assumes that $\rightarrow$ is acyclic, which ensures soundness by construction (by Lemma 6); again, no online consensus is needed.

In this section we examine the design space in order to understand the tradeoffs and possibilities in the spectrum.

4.5.1 Unsound combinations

One conflicting decision to avoid is creating a $\rightarrow$ cycle between guaranteed actions. For instance, consider a system consisting of four sites, with two noncommuting actions α and β. Suppose the four sites independently make the following decisions: Site 1 decides to guarantee α (by setting $\text{INIT} \triangleright \alpha$), Site 2 to guarantee β, Site 3 sets $\alpha \rightarrow \beta$ and Site 4 $\beta \rightarrow \alpha$. Any three of those four decisions are safe, but the four together violate mergeability. (The example also works with two or three sites.)

The second kind of conflict is to violate $\triangleright$-soundness. Consider actions γ and δ. If Site 1 makes γ guaranteed (e.g., $\text{INIT} \triangleright \gamma$), Site 2 makes δ dead (e.g.,

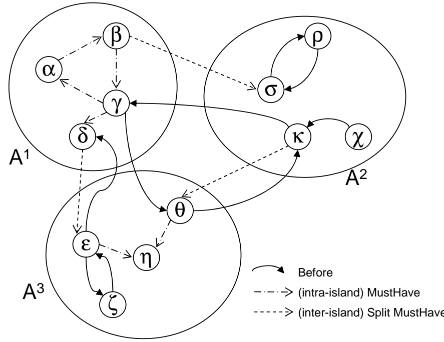

Figure 2: The system of Figure 1 partitioned into three islands.

$\delta \rightarrow \delta$), and Site 3 sets the dependency $\gamma \triangleright \delta$. Again, any two out of those three decisions remains sound, but not all three together.

If any of these unsound combinations can appear, then decision entails a consensus between all the concerned sites.

4.5.2 Safe decisions

Some decisions can be made unilaterally (locally) safely; these represent sufficient conditions for deciding safely. Assume for instance the only constraint on action α is $\alpha \triangleright \beta$. It is always safe to guarantee α, regardless of β. Conversely, if the only constraint on β is $\alpha \triangleright \beta$, then it is always safe to make β dead. Consider now a Before relation $\gamma \rightarrow \delta$. If (say) γ is not the target of another $\rightarrow$, then there is no cycle, and γ can be either guaranteed or made dead unilaterally; otherwise, the only safe unilateral decision is dead. Similarly for δ. If there is a $\rightarrow$ cycle (or, conservtively, if a cycle is suspected) then it is safe to make either action dead, or both. An ordering is sufficient to avoid unecessary aborts. When an action is involved in several constraints, a unilateral decision is safe only if safe with respect to all the constraints.

5 Partial replication

Shared data is often partitioned into separate databases. For instance, different subtrees of a shared file system [12]; or federated Web services each is a separate database. Databases are not necessarily independent; for instance, the root of a file-system subtree depends on its parent in a different database. A given site might replicate an arbitrary subset of the databases, and receive only the corresponding actions and constraints. Thus partial replication violates the eventual propagation properties. However, if Mergeability and Eventual Decision are both satisfied, the system retains important consistency properties in that schedules are sound, sites that do communicate may not make conflicting decisions, and sites can't "cheat" by leaving schedules under-determined.

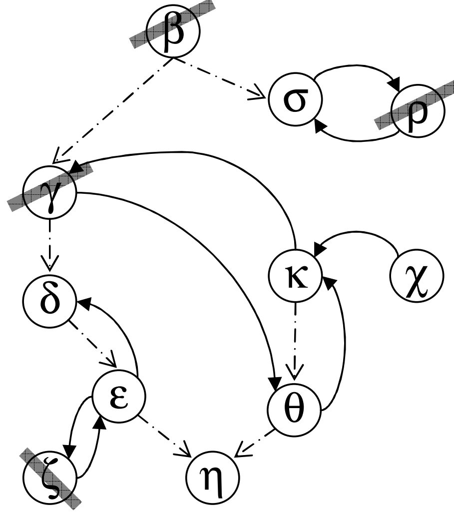

Figure 3: A possible execution of the algorithm of Section 5.2, proceeding from η upwards: greyed actions are dead, the others guaranteed. (α deleted from Figure 1 and layout rearranged.)

5.1 Islands, Split MustHave

Assuming that an action affects a single database, we partition actions into *islands*, as in Figure 2: $A = A^1 \uplus A^2 \uplus \ldots \uplus A^n$. $\mathit{Replicas}(d)$ denotes the set of sites replicating island d ($\forall d : |\mathit{Replicas}(d)| > 0$) and $\mathit{Islands}_i$ the set of islands replicated at site i. Figure 4 shows an example with three sites and their local multilogs.

We might define a new Eventual Partial Action Propagation property as follows: For any pair of such sites $i, j \in \mathit{Replicas}(d)$ and for any time t, there exists a time t' when $(K_i(t) \cap A^d) \subseteq (K_j(t') \cap A^d)$. Eventual Partial Constraint Propagation is similar.

Unfortunately $\triangleright$ is not adequate for partial replication, because if $\alpha \triangleright \beta$, then a site that executes α must also know β. Therefore we define a version that is "remotable" across islands, Split MustHave, noted $\triangleright\!\!\triangleright$. It behaves like $\triangleright$ when islands are replicated on the same site but remains meaningful when they are not. The results of the previous sections remain true when replacing $\triangleright$ with $\triangleright\!\!\triangleright$.

Definition 14 *The relation $\triangleright\!\!\triangleright \subseteq (A \cup \{\textsc{init}\}) \times (A \cup \{\textsc{init}\})$ is defined as follows. For any two actions $\alpha \in A^d$ and $\beta \in A^{d'}$, and for any site i and time t:*

$$\alpha \triangleright\!\!\triangleright \beta \stackrel{\text{def}}{=} \begin{cases} \textit{If } \{d, d'\} \subseteq \mathit{Islands}_i : \forall S \in \Sigma(M_i(t)), \alpha \in S \Rightarrow \beta \in S \\ \textit{Otherwise: } \exists i' : \alpha \in \mathit{Guar}(M_i(t)) \Rightarrow \beta \in \mathit{Guar}(M_{i'}(t)) \end{cases}$$

Practically, site i may guarantee α after it receives a message from i' that β is guaranteed.

site 1: M_1 site 2: M_2 site 3: M_3

Figure 4: Partial replication of the system of Figure 2. Site 1 replicates islands A^1 and A^2. Site 2 replicates islands A^2 and A^3 but has not yet received action κ. Site 3 only replicates A^2. None of the sites has yet received constraint $\gamma \rightarrow \theta$.

5.2 A decentralised decision algorithm

It would be nice to decide for each database separately (for instance, at a per-database primary), rather than via a system-wide consensus. We sketch a practical algorithm for this, due to Chong and Hamadi [3]. It can be implemented efficiently, assuming constraints are mainly due to the standard relations shown in Figure 1: parcel, causal dependence or mutual exclusion. Causal dependence is acyclic. Parcels are small, with occasional causality links between them that do not (in the expected case) create larger MustHave cycles. Because an island maps to a database, $\alpha \leftrightarrow \beta$ pairs are always within an island. Most $\rightarrow$ cycles are normally intra-island.

We present a simplified version of the algorithm that assumes the $\rhd\!\rhd$ graph is acyclic and makes unilateral safe decisions. The full version (to be published separately) tolerates occasional cycles of $\rhd\!\rhd$, which require a localised consensus.

Initially, assume there is a single primary site that replicates all the islands (we will relax this assumption shortly). This site runs the following centralised algorithm. All actions are initially undecided. The primary performs the following sequential algorithm, as illustrated in Figure 3. Given any chain $\alpha_n \rhd\!\rhd \ldots \rhd\!\rhd \alpha_1 \rhd\!\rhd \alpha_0$, the algorithm starts from the maximal element α_0 and works back to the minimal element.[6] Given some arbitrary pair $\alpha \rhd\!\rhd \beta$ where both α and β are undecided: if β is part of a cycle of $\rightarrow$ and all other actions in the cycle are guaranteed, then make both α and β dead (in order to maintain $\rightarrow$-soundness); otherwise guarantee β and leave α undecided. If β is now dead and there exists also $\gamma \rhd\!\rhd \beta$, then make γ dead (to maintain $\rhd$-soundness). The algorithm continues as long as there remains an undecided target of a MustHave. If an action MustHave two others ($\alpha \rhd\!\rhd \beta$ and $\alpha \rhd\!\rhd \gamma$ for instance) it is dead if either one is dead, and guaranteed if they are both guaranteed. Guaranteed actions related by $\rightarrow$ execute in that order, otherwise order is arbitrary. The algorithm terminates and maintains soundness, since it makes only safe unilateral decisions.

We now generalise to the case where different islands may have different

[6] An action unrelated to any other by $\rhd\!\rhd$ is considered a chain of length 0.

primaries. We run the following slightly modified algorithm at each primary. If $\rightarrow$ cycles are always intra-island, the algorithm does not change, and primaries may execute in parallel. For any relation $\alpha \rhd\!\!\rhd \beta$, before deciding on α, the primary of α waits to receive the decision from the primary of β. Since no $\alpha \nleftrightarrow \beta$ pair spans islands, primaries never need to agree on an ordering. Information flows in the opposite direction of the $\rhd\!\!\rhd$ graph, and no consensus is needed to decide.

In some applications, a $\rightarrow$ cycle may occasionally span islands. Then, the implementation must be careful to avoid concurrently guaranteeing the two last actions in a cycle, by ordering primaries (e.g., by IP address); when in doubt, the lowest-ordered primary in a cycle of $\rightarrow$ makes its action dead.

Our description assumed that the set of actions and constraints is fixed. The algorithm can be executed incrementally if actions and constraints are added according to causal order. Otherwise, a global algorithm will be needed to detect termination.

6 Modeling protocols in the action-constraint framework

In this section we study a number of published replication systems.

Many systems restrict actions to only reads and writes. Hereafter $r(x)$ denotes a read of object x; a write is noted $w(x)$.

6.1 Timestamp ordering (Last Writer Wins)

The Timestamp Ordering approach, also called the Last Writer Wins (LWW) algorithm or the Thomas Write Rule [18], is used in many practical systems, for instance distributed file systems [12, 21] or distributed directory systems [13]. Each site is free to read and write shared objects locally with no synchronisation. Objects and writes are timestamped, and a write only takes effect if its timestamp is greater than the object's. Writes propagate in arbitrary order but the system converges of timestamps. (Such systems don't propagate reads.)

Let us formalise LWW. Action α writes value $\alpha.value$ to the contents $x.contents$ of some object x, but only if the timestamp relation $\alpha.ts > x.ts$; otherwise it has no effect. Formally: $\alpha \stackrel{\text{def}}{=}$ if $\alpha.ts > x.ts$ then $x.contents :=$ $\alpha.value$ else *skip*. Somewhat surprisingly, under this definition, two writes to the same object commute.

Every action is guaranteed immediately when submitted. Since actions commute and are guaranteed, they are immediately decided and stable. There are no further constraints between actions. $\rightarrow$ is empty (hence trivially acyclic), and LWW satisfies Mergeability.

LWW is completely decentralised; there are no online decisions and no need to run a consensus algorithm. The drawback is that LWW loses information,

since when two writes occur concurrently the one with the lowest timestamp has no effect.

6.2 Causal Consistency

Up to now we have considered actions as opaque objects, submitted by a client external to the system. In practice the user cares about the context; for instance he might write out a check based on the knowledge that his account has sufficient balance.

More generally an action might depend on all previous history. Such application-specific dependencies are often unknown, and in any case are difficult to express. Therefore *causal consistency* conservatively assumes that an action depends on the current state. If actions are restricted to reads ($r_i(x)$ reads object x and is submitted at site i) and writes ($w_i(x)$), writes that might be causally related must execute at all processes in the same order; the order of concurrent writes is not determined. Then (1) $r_i(x)$ depends on the last $w_j(x)$ *executed* at i; (2) any action submitted at site i depends on all actions previously submitted at i; and (3) $w_k(x)$ submitted at $k \neq i$ cannot execute before $r_j(x)$ concurrently executed at i. Concurrently-submitted writes remain unordered.

Recall that a causal dependence $\alpha \rightsquigarrow \beta$ is encoded as $\alpha \rightarrow \beta \wedge \beta \rhd \alpha$. If neither $\alpha \rightsquigarrow \beta$ nor $\beta \rightsquigarrow \alpha$ then α and β are said concurrent.

A causally-consistent system is characterised by the following rule. We will subscript actions with their submission site and time.

Transition Rule 2 (Causal Consistency) *Consider some actions $\alpha, \beta, \dots$, classified as reads noted $r(x)$ and writes noted $w(y)$. Note the site where α is submitted $\mathrm{ss}(\alpha)$ and the time at which it is submitted $\mathrm{st}(\alpha)$. Note $S_i(t)$ the schedule at site i at time t.*

1. *An action depends causally on all actions previously submitted at the same site: $\forall \alpha, \beta : \mathrm{st}(\alpha) < \mathrm{st}(\beta) \Rightarrow \alpha \rightsquigarrow \beta$.*

2. *A read action $\alpha = r(x)$ depends causally on the last write to x executed at the same site: $\beta \in S_{\mathrm{ss}(\alpha)}(\mathrm{st}(\alpha)) \wedge (\gamma \in S_i(t) \Rightarrow \gamma <_{S_i(t)} \beta \vee \beta = \gamma) \implies w(x) \rightsquigarrow r(x)$, where β and γ are both writes to x.*

Note that it probably doesn't make much sense to submit an action until the ones it depends upon are stable.

Causal consistency is a decentralised decision algorithm that does not lose information. However, despite its name, it is not a consistent according to our definitions, because concurrent writes to the same object remain unordered [17], despite not commuting. Thus, causal consistency does not satisfy the Eventual Decision property.

6.3 Commutative ESDS

This section describes the commutative version of the Eventually Serialisable Data Service protocol [7, Section 4.3]. In this protocol, which we call C-ESDS,

operations are assumed to commute, except when an operation α explicitly indicates causal dependencies on a set of operations $\alpha.prev$. C-ESDS ensures that the schedules executed at each replica obey these dependencies, and replicas eventually converge. The order in which actions are scheduled is determined by *labels* that are assigned, in causal order, to guaranteed actions. There are no conflicts, and every submitted action is guaranteed.

The causal dependencies between operations can be translated into our notation: for all $\beta \in \alpha.prev : \beta \rightarrow \alpha \wedge \alpha \rhd \beta$. They are assumed acyclic by construction.

The transition rules for C-ESDS are the following (to simplify notation we omit the site and time parameters, i.e., K stands for $K_i(t)$, and similarly for $\rightarrow$ and $\rhd$):

Transition Rule 3 (Commutative ESDS)

1. *Every action is guaranteed:* $\forall \alpha \in A : \alpha \in K \Rightarrow \text{INIT} \rhd \alpha$.
2. *Causality constraints:* $\forall \alpha, \beta \in A : \alpha \in K \Rightarrow (\beta \in \alpha.prev \Rightarrow (\beta \rightarrow \alpha \wedge \alpha \rhd \beta))$.
3. *Causality is acyclic:* $\nexists \alpha_1, \ldots, \alpha_n \in K : \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_1$.
4. *Non-commuting actions are serialised by the client:* $\forall \alpha, \beta \in A, \alpha \not\leftrightarrow \beta \wedge \alpha \in K \wedge \beta \in K \Rightarrow (\alpha \rightarrow \beta \vee \beta \rightarrow \alpha)$.
5. *Causal dependence is eventually satisfied:* $\forall \alpha, \beta \in A : \alpha \in K_i(t) \wedge \beta \in \alpha.prev \Rightarrow \exists t' : \beta \in K_i(t')$.

The only constraints are those allowed by items 1 and 2; in particular item 4 does not introduce new constraints. The above is in addition to the Universal Rule 1.

Thus formalised, it is clear that C-ESDS satisfies consistency, by Uniform Local Soundness. Every action is guaranteed and $\rightarrow$ is acyclic, hence all multi-logs are sound. By item 5, every action is eventually decided.

Reasoning within our model clarifies the specification of C-ESDS and suggests two simplifications to the protocol specified by Fekete et al. First, an action is stable as soon as its antecedents are guaranteed at the local site; it is not necessary to wait, as they do, for the action to be stable at other sites. Second, all local orders compatible with the *prev* constraints are equivalent, and the global total order they compute is unnecessary.

6.4 Non-Commutative ESDS

In the full ESDS system of Fekete *et al.* [7], which we call NC-ESDS, actions do not commute, but the *prev* lists submitted by clients provide only a partial ordering. The protocol computes a total ordering, ensuring actions are executed in the same order at every site.

The total order is computed as follows. Every site assigns every known action α a *label* (a timestamp), as soon as all actions in $\alpha.prev$ have themselves been labeled. The labelling algorithm ensures that, for all β that already has a label (which includes every action in $\alpha.prev$), the label of α is greater. The location of α in the total order is given by the minimum label assigned to it by any site; this is known at site i once i has received notification of α's label assignment from every other site.

The multilogs of Fekete *et al.* are set up differently from ours. The multilog at site i contains: the actions known at i, the set of actions that i knows are "done" (i.e., they have received a label) at every site j, and the set of actions that i knows are "solid" (i.e., their minimum label is known) at every site j.

If we assume that communication is live, and for any known action α all actions in $\alpha.prev$ are eventually known, then α will eventually become "done" everywhere. Once site i observes that action α is "done" everywhere, it can set $\beta \rightarrow \alpha$ for any action β that is "done" everywhere and where the minimum label of β is less than the minimum label of α. Every site will eventually make the same computation, and the constraint $\beta \rightarrow \alpha$ will be known at every site. Thus, every action is eventually serialised, and the serialisation order is the same at every site; mergeability is satisfied. Furthermore, every action is guaranteed as soon as it is submitted. Eventual decision is satisfied, and every action is eventually stable.

The "solid" set is not strictly necessary. Fekete *et al.* use it to be able to observe when an action has become stable.

We plan to formalise the above analysis of NC-ESDS in future work.

6.5 Transactions

In this section we examine how to represent conventional serialisable transactions.

A transaction is a group of actions with the ACID properties:

- Atomic or all-or-nothing: either all its actions *commit* or they all *abort*. In our model, mutual $\triangleright$ relations ensure atomicity. To abort is equivalent to becoming dead. To commit is to become stable and guaranteed.
- Correct (executed alone, the transaction maintains the application invariants). This is an assumption.
- Isolated (no committed transaction can observe intermediate effects of another transaction).
- Durable (the effects of a committed transaction are persistent). Durability is captured by our stability concept.

The classical correctness property is *serialisability*: committed schedules are all equivalent to some sequential ordering of transactions.

In the literature, a transaction's actions are bracketed by special begin/end markers. A pessimitic transaction system decides the serialisation order transaction begin. Deferred-update systems [1] serialise when the current transaction ends; but when a cycle of $\rightarrow$ is observed (or suspected) transactions abort (are

made dead). A system such as Bayou [22, 14] allows chains of outstanding transactions (and support unlimited rollback). Related transactions are sent to a "primary" site to be decided in bulk. In Bayou, the serialisation is known *a priori* using timestamps, but transactions may abort because of hidden dependencies.

In this section, we follow the classic usage whereby actions are restricted to reads or writes to shared objects, e.g., $r_i(x)$ to read object x or $w_i(x)$ to write it.

6.5.1 Action-Constraint representation of transactions

In our formalisation, we use a single transaction marker. We do not preclude cascading aborts nor concurrency within transactions. Stronger assumptions are easily represented by adding appropriate constraints.

In this section, indices denote transaction identity. A transaction T_i is composed of a single "transaction marker" action τ_i and zero or more read and write actions α_i. To ensure atomicity, for all actions α_i of transaction T_i the following are mandatory: $\alpha_i \rhd \tau_i$ and $\tau_i \rhd \alpha_i$.

A τ_i action does not change the state. Transaction markers are mutually non-commuting if one transaction writes an object that the other one reads or writes: $\tau_i \not\leftrightarrow \tau_j$ if $i \neq j \wedge (R_i \cap W_j \neq \emptyset \vee W_i \cap R_j \neq \emptyset \vee W_i \cap W_j \neq \emptyset)$, where R_i is T_i's read set (the set of x such that $r_i(x)$) and W_i is T_i's write set (the set of x such that $w_i(x)$).

A write to some object is non-commuting with both reads and writes to the same object. For all read and write actions in transactions T_i, T_j upon all objects x, y:

- $r_i(x) \not\leftrightarrow w_j(y)$ iff $x = y$
- $w_i(x) \not\leftrightarrow w_j(y)$ iff $x = y$

The following rule ensures isolation.

Transition Rule 4 *If T_i is serialised before T_j (i.e., $\tau_i \rightarrow \tau_j$) then their non-commuting actions must be serialised in the same order.* $\tau_i \rightarrow \tau_j \iff (\alpha_i \not\leftrightarrow \beta_j \Rightarrow \alpha_i \rightarrow \beta_j)$*, where α_i and β_j are arbitrary actions of T_i and T_j respectively.*

A transaction system evolves over time according to the conjunction of the general transition rules of Section 4.2 and the specific Transition Rule 4.

The rule ensures transaction isolation, because if $\tau_i \rightarrow \tau_j$, every action of T_i executes before any action of T_j, unless they commute.

Theorem 11 (Rule 4 ensures isolation) *Consider a transaction system where T_i and T_j may read or write shared objects x, y, etc. Assume the system follows the Transaction Isolation Rule. Then, if $\tau_i \rightarrow \tau_j$, all reads and writes of x (resp. y, etc.) by T_j follow any write of x (resp. y, etc.) by T_i.*

The proof is by inspection.

In this context, serialisability is obtained if the graph of $\rightarrow$ relations between guaranteed τ_i actions is acyclic. This is the same as saying that multilogs are

sound. Thus, our Mergeability property is equivalent to the classical Serialisability.

Theorem 12 *Consider a replicated transactional system that obeys Rule 4. Transactions are serialisable if and only if the system satisfies Mergeability.*

The proof derives from Lemma 6.

6.5.2 Causal dependence between transactions

In the above model, transactions are essentially independent. In practice this might not be true: for instance, the user might submit a transaction to reserve a hotel based on the knowledge that a previous transaction booked a flight. Such application-specific dependencies are often unknown and in any case are difficult to express. Therefore, many authors posit causal dependencies between successive transactions [4]: e.g., if T_j reads object x that was last written by T_i, there is a causal dependence $w_i(x) \rightsquigarrow r_j(x)$.[7] Furthermore, a read cannot depend on a write that is not serialised before it. Hence the following transition rule:

Transition Rule 5 (Causally dependent transactions) *Consider transactions $T_i, T_j, \ldots$ composed of reads ($r_i(x)$), writes ($w_i(x)$) and transaction markers (τ_i). Note $S_i(t)$ the schedule at site i at time t.*

1. *A read action $r_i(x)$ submitted at time t depends causally on the last write to x executed at Site i: $w_j(x) \in S_i(t) \wedge (w_k(x) \in S_i(t) \Rightarrow (w_k(x) <_{S_i(t)} w_j(x) \vee k = j)) \implies w_j(x) \rightsquigarrow r_i(x)$*
2. *A read action $r_i(x)$ submitted at time t must be serialised before any concurrent or later $w_j(x)$. $w_j(x) \notin S_i(t) \Rightarrow r_i(x) \rightarrow w_j(x)$.*

It makes sense to submit a new transaction only when the ones it causally depends upon are stable. In the example above, this would avoid the user booking the hotel only to later discover that the flight reservation has failed.

Transactions systems with causal dependence follow the Universal Transistion Rule 1, the Transaction Isolation Rule (Rule 4) and a Causal Dependence Rule [19] . Transactions are ordered according to the (acyclic) causal dependency. Bullet 2 creates a $\rightarrow$ cycle between concurrent transactions updating the same object, so at least one of them must be aborted in order to satisfy Mergeability.

6.6 Holliday's partial replication protocol

We now describe the partial replication protocol, for transactional systems, of Holliday *et al.* [11]. We focus on its partial replication and consistency properties.

The authors assume FIFO communication with eventual action and constraint propagation. They implement a vector-clock mechanism that enables

[7] Recall that $\alpha \rightsquigarrow \beta$ is shorthand for $\alpha \rightarrow \beta \wedge \beta \rhd \alpha$.

some site to know that another site has observed a given event. Abort/commit decisions are taken by the *pre-commit* consensus algorithm among the sites that replicate the databases accessed by the transaction.

We augment multilogs with a new set of actions, X, containing markers for transactions that have been proposed for pre-commit. We assume that like K, X grows monotonically and its contents are eventually propagated to all sites. We posit a primitive $\text{obs}_j(\tau, i)$ that returns true when site j knows (thanks to the vector clock) that site i has executed the Abort Decision step, relative to τ, of Transition Rule 6.

Transition Rule 6 (Holliday's pre-commit algorithm)

1. Transaction termination: *At site i where transaction T is submitted, when T terminates, start the pre-commit protocol by inserting its marker τ into the local X set: $X_i := X_i \cup \{\tau\}$.*
2. Abort decision: *At any site j, check transaction markers τ in X whether any marker τ' known locally is non-commuting with τ; if so, abort both T and T'. Formally: $\forall \tau \in X_j, \forall \tau' \in K_j$ such that $\tau \not\leftrightarrow \tau'$: $M_j := M_j \cup \{\tau \to \tau, \tau' \to \tau'\}$. Henceforth, other sites k will eventually observe $\text{obs}_k(\tau, j) = \text{true}$.*
3. Commit decision: *When site i where T was submitted knows that every site has executed the Abort Decision step for τ, and none of them has aborted T, then make T committed. Formally: $\forall \tau \in X_i$: if $\forall k\ \text{obs}_i(\tau, k) \wedge \neg(\tau \to \tau)$ then $M_i := M_i \cup \{\text{INIT} \rhd \tau\}$. (Holliday assumes FIFO communication.)*

Transactions are ordered by the causality relation, which is assumed acyclic. If two concurrent transactions are non-commuting, then both are made dead in the Abort Decision step above. A transaction is allowed to become guaranteed (in the Commit Decision step) only if it does not abort. No action is both dead and guaranteed and $\to$ is acyclic. Since every action is eventually decided, and we assumed eventual propagation of both actions and constraints, Holliday's protocol is consistent.

Our simplified specification makes it easier to understand the protocol than the original informal description. It suggests a possible improvement in Abort Decision step: to use a total order of transactions (e.g., by numerical transaction identifier) and abort only the lowest-ordered one.

7 Related work

Our survey of optimistic replication [18] motivated us to understand the commonalities and differences between protocols. The relations between consistency and ordering have been well studied in the context the causal dependence relation [16, 17]. We believe our simpler and modular primitives clarify and generalise the analysis. The primitives are common to all protocols, as are the significant events of actions becoming guaranteed, dead, serialised, decided and stable. We are able to specify formally a large spectrum of systems and algorithms.

Much formal work on consistency focuses on serialisability [1, 5]. Mergeability subsumes serialisability and applies to a larger spectrum of systems.

Our approach is not directly comparable to linearisability [10], as our schedules execute actions in isolation and because we abstract away from objects and from real time, which both are part of the definition of linearisability.

Constraints $\rightarrow$ and $\triangleright$ were first proposed by Fages [6] for general reconciliation problems in optimistic replication systems.

Our approach has many similarities with the Acta framework [5]. Acta has been used to compare a number of database protocols [16]. Acta provides a set of logical primitives over execution histories, including presence of an event, implication, and causal dependence and ordering between events. Acta makes assumptions specific to databases, such as the existence of transaction commit and abort primitives. The Acta description language is more powerful and is used to analyze protocols at a much finer granularity. On the other hand, the action-constraint language is simpler; it is straightforward to translate most of the Acta dependencies into our language.

8 Conclusions and future work

We have presented a novel approach to the description of replication protocols and their consistency properties. Actions (operations accessing shared data, submitted by clients of a replicated system) are connected by binary constraints, which are invariants that must be maintained by the system, and which specify legal schedules. Different entities such as users, application, shared objects, and replication protocols, contribute their own constraints. The focus of this work is to describe replication protocols, which propagate actions and constraints between sites, and add sufficient constraints to ensure consistency.

We were able to describe very different replication protocols in the same language and to bring to light their commonalities and differences. We identified some primitive components common to all protocols (the significant sets). We provide a generic formulation of the consistency-related properties of protocols. Several definitions of consistency were shown to be equivalent, underscoring that, although diverse, many protocols are in a deep sense equivalent. Protocols differ from one another by the kinds of constraints that they support and by the transition rules that govern evolution of the system over time.

We defined a new safety condition for consistency, called Mergeability, which generalises serialisability. It makes it clear that in the general case, consistency entails a global synchronisation, which is known to be difficult. However we have shown sufficient conditions (acyclic Before graph, and safe unilateral decisions) where it is possible to decide locally and incrementally. In contrast to the other conditions, Mergeability is appropriate for partially replicated systems. We described a new protocol based on this insight.

We are aware of a number of limitations of this work. The current version of the theory totally ignores fault tolerance aspects. The split between constraints and transition rules is somewhat unnatural. The algorithm leading up to a

decision is outside of the theory, which is somewhat unsatisfactory.

The biggest limitation is the weakness of the constraint language. There are only two kinds of constraints, both binary. This has the advantages of simplicity (it is easy to understand and to prove properties), and is sufficient to describe a number of diverse systems. However practical systems need to enforce invariants that cannot be described in the current language, for instance the invariant that account balances must remain positive in a banking application. A possible direction is go generalise our guaranteed and dead sets to guaranteed and dead patterns. Currently, the crucial safety property is that the guaranteed and dead sets are globally disjoint. This would generalise to the guaranteed and dead sublanguages be disjoint globally.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. http://research.microsoft.com/pubs/ccontrol/.

[2] A. D. Birell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25:260–274, April 1982.

[3] Yek Chong and Youssef Hamadi. Distributed IceCube. Private communication, January 2004.

[4] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: The SAGA continues. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 10, pages 349–397. Morgan Kaufmann, 1992.

[5] Panos K. Chrysanthis and Krithi Ramamritham. Correctness criteria and concurrency control. In A. Sheth, A. K. Elmagarmid, and M. Rusinkiewicz, editors, *Management of Heterogeneous and Autonomous Database Systems*, chapter 10. Morgan-Kaufmann, 1998. http://www-ccs.cs.umass.edu/db/publications/mdb.ps.

[6] François Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *6th Annual W. of the ERCIM Working Group on Constraints*, Prague, Czech Republic, June 2001.

[7] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Conf. on Principles of Dist. Comp.*, Philadelphia PA, USA, May 1996. http://theory.lcs.mit.edu/~alex/podc96esd.ps.

[8] M. Fisher, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):274–382, April 1985.

[9] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Francisco CA, USA, 1993. ISBN 1-55860-190-2.

[10] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[11] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Partial database replication using epidemic communication. In *22th Int. Conf. on Distr. Comp. Sys. (ICDCS)*, pages 485–493, Vienna, Austria, July 2002. IEEE Computer Society. http://computer.org/proceedings/icdcs/1585/1585toc.htm.

[12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992. http://www.acm.org/pubs/contents/journals/tocs/1992-10.

[13] Microsoft. *Windows 2000 Server: Distributed Systems Guide*, chapter 6, pages 299–340. Microsoft Press, Redmond, WA, USA, 2000.

[14] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symp. on Operating Systems Principles (SOSP-16)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. http://www.parc.xerox.com/csl/projects/bayou/.

[15] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, Catania, Sicily, Italy, November 2003.

[16] Krithi Ramamritham and Panos K. Chrysanthis. A taxonomy of correctness criteria in database applications. *VLDB Journal*, 5(1):85–97, 1996.

[17] Michel Raynal and Masaaki Mizuno. How to find his way in the jungle of consistency criteria for distributed shared memories (or how to escape from Minos' labyrinth). In *Proc. of the IEEE Int. Conf. on Future Trends of Distributed Computing Systems*, pages 340–346, Lisboa (Portugal), September 1993.

[18] Yasushi Saito and Marc Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, October 2003. http://research.microsoft.com/research/pubs/view.aspx?tr_id=681.

[19] Marc Shapiro and Karthik Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, March 2004. Draft available from http://www-sor.inria.fr/~shapiro/tmp/marc.ps.gz.

[20] Marc Shapiro, Nuno Preguiça, and James O'Brien. Rufis: mobile data sharing using a generic constraint-oriented reconciler. In *Conf. on Mobile Data Management*, Berkeley, CA, USA, January 2004. http://www-sor.inria.fr/~shapiro/papers/mdm-2004-final.ps.gz.

[21] Hal Stern, Mike Eisley, and Ricardo Labiaga. *Managing NFS and NIS*. O'Reilly & Associates, 2nd edition, July 2001. ISBN 1-56592-510-6.

[22] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), December 1995. ACM SIGOPS. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.