A Constraint-based Formalism for Consistency in Replicated Systems

Marc Shapiro, Karthikeyan Bhargavan, Nishith Krishna*
Microsoft Research, Cambridge
7 JJ Thomson Ave, Cambridge CB3 0FB, United Kingdom; marc.shapiro@acm.org

Abstract

We present a formalism for modeling replication in a distributed system with concurrent users sharing information. It is based on actions, which represent operations requested by independent users, and constraints, representing scheduling relations between actions. The formalism encompasses semantics of shared data, such as commutativity or conflict between actions, and user intents such as causal dependence or atomicity. It enables us to reason about the consistency properties of a replication protocol or of classes of protocols. It supports weak consistency (optimistic protocols) as well as the stronger pessimistic protocols. Our approach clarifies the requirements and assumptions common to all replication systems. We are able to prove a number of common properties. For instance consistency properties that appear different operationally are proved equivalent under suitable liveness assumptions. The formalism enables us to design a new, generalised peer-to-peer consistency protocol.

Keywords Consistency, models for distributed systems, replicated data, optimistic replication, semantics of shared data, weak consistency, replication protocols.

1 Introduction

Replicating data in a distributed system improves availability at the cost of maintaining consistency, since each site's view may be partial or stale. It is well accepted that replication can be made more efficient by taking semantics into account, but it is difficult to reason about the correctness of such weaker protocols. Partial replication constitutes an further complication. Despite a large body of previous work [15], we lack a formal framework for understanding, reasoning about, and comparing replication protocols. This paper presents such a framework.

We model a distributed system as a replicated database and a replication protocol. Users independently submit queries and updates to the database, abstracted as *actions*. End users, applications and data types (together abstracted as clients in this framework) also submit scheduling *constraints* to express important intents or semantic properties.

Each site has a local view (called multilog) of known actions and constraints. The site executes a schedule, which completely determines the state of the replica. Sites converge if they execute the same actions in the same order, which a replication protocol ensures, if necessary, by adding more constraints.

Our contributions are the following. We propose a novel framework for reasoning about replicated systems. It is the first that unifies: data semantics such as commutativity and conflicts, application semantics such as causal dependence, user intents such as atomicity, and protocol decisions about which operations to execute and in which order. Our framework is simple, and gives a semantics of replication in terms of constrained sets.

Our results clarify the requirements and assumptions of a replication system. As an example, we will model different systems in our framework, e.g., Bayou and ESDS, and prove them consistent.

We are able to prove interesting properties for classes of replication protocols. For instance, we identify four different notions of consistency, which seem to dif-

*New York University

fer in their operational requirements. We are able to prove that, under sufficiently strong liveness assumptions, they are equivalent.

The framework can guide the design of new replication protocols. We propose a new distributed replication algorithm generalising Bayou, whose design is directly guided by the framework. It shows that constraints can be used as an implementation mechanism as well as specification framework.

The paper proceeds as follows. Section 2 overviews the basic formalism. Section 3 defines and compares consistency properties. We examine some decision algorithms from the literature and rules for local decision in Section 4. We derive a novel decentralised replication algorithm in Section 5. Section 6 compares with related work, and we conclude in Section 7 with a summary of contributions and future work.

A separate technical report [16] provides a complete formal treatment. Here we focus on presenting the intuitions and keep the formalism to a minimum.

2 Formal framework

Each site in a replicated system maintains a local view called *multilog*.[1] The current state results from executing a *sound* (i.e., valid) schedule computed from the multilog. Over time the multilog grows (and conceptually never shrinks) by addition of actions and constraints, either *submitted* by local clients or received from remote sites. The set of sound schedules grows with the number of actions and shrinks as the number of constraints increases.

2.1 Actions and schedules

Slightly more formally, A is the set of unique actions $\text{INIT}, \alpha, \beta, \ldots$. Actions are assumed deterministic[2] but are otherwise uninterpreted. The *non-action* $\overline{\alpha}$ is a placeholder with no effect (non-actions will be useful when discussing liveness). Action INIT represents the initial state and has no effect.

[1] We call it a multilog and not a log, because it contains actions submitted at several sites and the actions are not ordered.

[2] Executing the same action from two equivalent input states yields equivalent output states.

A *schedule* is a non-empty sequence of actions and non-actions, for instance $S = \text{INIT}.\alpha.\overline{\beta}.\gamma$. In this example, α is *executed* (noted $\alpha \in S$), and β is non-executed (noted $\overline{\beta} \in S$); all four actions are said *scheduled* (noted $sched(\alpha, S)$). A given action may appear only once in a schedule, either as executed or as non-executed. The ordering is noted $<_S$. Every schedule starts with INIT. Intuitively, a non-action in a schedule indicates that the scheduler is aware of the action but does not execute it, e.g., because of a constraint.

Actions commute unless specified otherwise by the notation $\alpha \leftrightarrow \beta$ (read "non-commuting"). A non-action commutes with every action and non-action. Two schedules are *equivalent* ($S_1 \equiv S_2$) if they execute the same actions, and non-commuting pairs of actions execute in the same order.

Commutativity allows us to model a number of real-world cases of schedule equivalence:

- Classically, actions commute if both are reads, or if they access independent variables.
- Overwriting: in some systems an out-of-order write has no effect; then writes effectively commute. For instance in timestamped replication (Last Writer Wins) [15], writing a file tests whether the write timestamp is greater than the file's; if so the write takes effect, otherwise it is a no-op [16].
- Reconciliation: An example of a reconciliation algorithm is Operational Transformation [20, 22]. Two actions submitted concurrently execute in arbitrary order. The second one to execute is transformed to ignore the effect of the first, in effect rendering them commutative.
- Failure or aborts: An action that fails or aborts becomes *dead*, i.e., appears as a non-action in all schedules, which commutes with all actions.

2.2 Multilogs and sound schedules

Multilog $M = (K, \rightarrow, \triangleright)$ represents a site's view. K is the set of known actions ($K \subseteq A$); $\rightarrow$ and $\triangleright$ are the set of known constraints. The relation $\rightarrow \subseteq A \times A$ (pronounced Before) is not necessarily acyclic, nor reflexive, nor transitive. Relation $\triangleright \subseteq A \times A$ (pronounced MustHave) is transitive and reflexive. By convention,

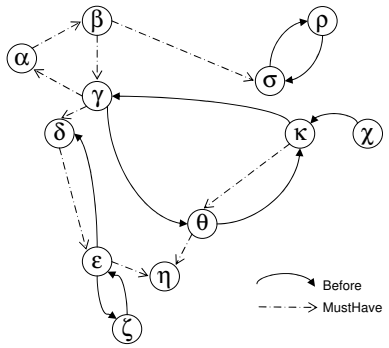

Figure 1. Example constraints. α, β and γ form a *parcel*, an atomic (i.e., all-or-nothing) execution. γ executes only if δ also executes. δ is *causally dependent* on ϵ. ϵ and ζ *conflict* with (i.e., mutually exclude) each other. Only two actions out of the three γ, θ and κ can execute. If both χ and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

t	0	1	2	3	4
$K_i(t)$	Ø	α	α, β	α, β	α, β
$\rightarrow_{i,(t)}$	Ø	Ø	Ø	$\alpha \rightarrow \beta$	$\alpha \rightarrow \beta, \beta \rightarrow \alpha$
$\rhd_{i,(t)}$	Ø	Ø	$\beta \rhd \alpha$	INIT $\rhd \beta, \beta \rhd \alpha$	INIT $\rhd \beta, \beta \rhd \alpha$
$\Sigma(M_i(t))$	INIT	INIT.$\overline{\alpha}$	INIT.$\overline{\alpha}.\overline{\beta}$	INIT.$\alpha.\beta$	Ø
		INIT.α	INIT.$\alpha.\overline{\beta}$		
			INIT.$\alpha.\beta$		
			INIT.$\beta.\alpha$		
$K_j(t)$	Ø	β	α, β	α, β	α, β
$\rightarrow_{j,(t)}$	Ø	Ø	Ø	$\beta \rightarrow \alpha$	$\beta \rightarrow \alpha, \beta \rightarrow \alpha$
$\rhd_{j,(t)}$	Ø	$\beta \rhd \alpha$	$\beta \rhd \alpha$	$\beta \rhd \alpha$	INIT $\rhd \beta, \beta \rhd \alpha$
$\Sigma(M_j(t))$	INIT	INIT.$\overline{\beta}$	same as i	INIT.$\overline{\alpha}.\overline{\beta}$	Ø
				INIT.$\alpha.\overline{\beta}$	
				INIT.$\beta.\alpha$	

Figure 2. Example execution with two sites i and j, and actions $A = \{\text{INIT}, \alpha, \beta\}$ where $\alpha \leftrightarrow \beta$. The two multilogs start empty (Time 0); the only possible schedule is INIT. • At Time 1, the multilogs receive actions and constraints (either from a client at their site, or from a third site). The views of the two sites are different, and so are the sound schedules each site can choose from. Note that the only sound schedule at Site j is INIT.$\overline{\beta}$ because $\beta \rhd \alpha$ but j does not know α. • At Time 2, the two sites have communicated and have identical multilogs and identical sets of sound schedules. Note however that the two sites have not converged, as they may execute non-equivalent schedules; for instance i might execute INIT.$\alpha.\beta$ while j could choose INIT.$\beta.\alpha$. • At time 3, Site i has ensured that both α and β must execute. Both sites have serialised the non-commuting actions. • At time 4 they have exchanged their multilogs. Now the system is unsound (there are no sound schedules) because the two sites chose opposite serialisation orders at Time 3. • If, say, only $\beta \rightarrow \alpha$ had been submitted at Time 3, then at Time 4 they would converge to schedule INIT.$\beta.\alpha$. Alternatively, if Site j had not submitted $\beta \rhd \alpha$ at Time 1, now they would converge to INIT.$\overline{\alpha}.\beta$.

for any $\alpha \in A$, INIT $\rightarrow \alpha$ and $\alpha \rhd$ INIT; this is left implicit in the rest of the paper.

Figure 1 gives some examples of constraints and of common combinations. Intuitively, $\alpha \rightarrow \beta$ indicates that a scheduler must maintain an ordering between the two actions: no schedule may execute β before α. A schedule that executes neither α nor β, or only α, or only β, or both α and β in that order (but not necessarily adjacent) is correct with respect to this constraint. Relation $\alpha \rhd \beta$ is an implication: if α executes in a schedule, then β must also execute somewhere in the same schedule, although not necessarily in that order. A schedule that executes only β, or that executes neither α nor β, is correct with respect to this constraint. Conversely, if the schedule non-executes β, then α may not execute.

The set of sound schedules of M is noted $\Sigma(M)$; M is said sound if $\Sigma(M) \neq \emptyset$. Schedule $S \in \Sigma(M)$ iff:

- Every action in K is either executed or non-executed in S: $\alpha \in K \Rightarrow sched(\alpha, S)$.
- Actions that execute in S are ordered by $\rightarrow$: $\alpha, \beta \in S \wedge \alpha \rightarrow \beta \Rightarrow \alpha <_S \beta$.
- MustHave behaves like implication: $\alpha \in S \wedge \alpha \rhd \beta \Rightarrow \beta \in S$.

For instance, the multilogs $M1 = (\{\alpha\}, \emptyset, \{\text{INIT} \rhd \alpha\})$ and $M2 = (\{\alpha\}, \{\alpha \rightarrow \alpha\}, \emptyset)$ are both sound. Their sound schedules are $\Sigma(M1) = \{\text{INIT}.\alpha\}$ and $\Sigma(M2) = \{\text{INIT}.\overline{\alpha}\}$. Their union $M3 = (\{\alpha\}, \{\alpha \rightarrow \alpha\}, \{\text{INIT} \rhd \alpha\})$ is not sound. Although it contains a $\rightarrow$ cycle, multilog $M4 = (\{\alpha, \beta\}, \{\alpha \rightarrow \beta, \beta \rightarrow \alpha\}, \emptyset)$ is sound, since $\Sigma(M4) = \{\text{INIT}.\overline{\alpha}.\overline{\beta}, \text{INIT}.\alpha.\overline{\beta}, \text{INIT}.\overline{\alpha}.\beta\}$. (We do not need to consider the sound schedules INIT.$\overline{\beta}.\alpha$ and INIT.$\beta.\overline{\alpha}$ since they are equivalent to the previous ones.)

We say that two multilogs are equivalent if they generate the same set of sound schedules: $M_1 \equiv M_2$ iff $\Sigma(M_1) = \Sigma(M_2)$. Note that $\Sigma(M)$ is closed with respect to schedule equivalence. Hereafter, we identify a multilog with its equivalence class.

This limited constraint language is surprisingly expres-

sive. We have used it to express the semantics of applications as diverse as a shared calendar, a travel reservation system and a replicated file system [12, 18]. For instance if α creates a directory and β a file in that same directory, the file system submits $\beta \rhd \alpha \wedge \alpha \rightarrow \beta$ (causal dependence) along with β.

A set of actions c is said to *conflict* if the actions in c form a $\rightarrow$ cycle. Intuitively, this means that no sound schedule can execute all the actions in c. For example, if $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$, then α and β conflict, i.e., there can be no sound schedule that executes both of them.

Figure 2 provides an example of scheduling in two different sites, as actions and constraints are submitted and propagated between sites.

2.3 Significant subsets and events of a replication protocol

Execution strategies vary widely between replication protocols: in some, actions execute immediately, in others they are deferred; execution order may be pre-established or computed; actions might roll back. However a protocol would be useless if it did not reach some final decision for every action. We represent decisions as constraints; the following *significant subsets* capture the possible stages of irrevocable decision:

- **Guaranteed** actions execute in every schedule. $Guar(M)$ is the smallest set satisfying: (1) INIT $\in Guar(M)$. (2) $\forall \beta \in A$: If $\alpha \in Guar(M)$ and $\alpha \rhd \beta$ then $\beta \in Guar(M)$.
- **Dead** actions non-execute in every schedule. $Dead(M)$ 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 $\alpha \in Dead(M)$.
- A **serialised** action is one that is ordered with respect to all non-commuting actions that execute. $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in A | \forall \beta \in A, \alpha \leftrightarrow \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M)\}$
- An action is **decided** once it is either dead, or both guaranteed and serialised. $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$
- An action is **stable** when its effects cannot change, i.e., it is either dead, or it is guaranteed and serialised and all preceding actions are themselves stable. (In practice, stable actions can be pruned from multilogs.) $Stable(M)$ is the smallest set satisfying: (1) INIT $\in Stable(M)$, (2) $Dead(M) \subseteq Stable(M)$, (3) If $(\alpha \in Guar(M) \cap Serialised(M)) \wedge (\forall \beta \in A : \beta \rightarrow \alpha \Rightarrow \beta \in Stable(M))$ then $\alpha \in Stable(M)$.

Note that if M is sound, every guaranteed action must be known: $Guar(M) \subseteq K$. Also note that $\alpha \rightarrow \alpha \Rightarrow \alpha \in Dead(M)$ and that INIT $\rhd \alpha \Rightarrow \alpha \in Guar(M)$. M is sound iff the guaranteed and dead sets are disjoint.

3 Replication and consistency

In this section, we describe liveness and safety properties that we require of replication systems, stated in terms of our action-constraint framework.

3.1 Site schedules and transition rules

Different replication systems (such as ESDS and Bayou) differ by the actions and constraints they accept, and by the decisions they make. We summarise a replication protocol by rules describing how the system changes from time t to $t+1$.

The current state of site i is the result of running a *site schedule* $S_i(t) \in \Sigma(M_i(t))$. In our framework, if $|\Sigma(M_i(t))| > 1$, then the choice between sound schedules is irrelevant for consistency, although individual replication systems may carefully pick a schedule for optimality.

Each site i has its own view $M_i(t) = (K_i, \rightarrow_i, \rhd_i)(t)$, evolving over time t, called its *site multilog*.[3] Multilogs are monotonically non-shrinking, which implies that the significant subsets of Section 2.3 are non-shrinking, and that an unsound multilog remains unsound forever.

All protocols obey a Universal Transition Rule, which says simply that a site may receive actions and constraints from a local client or from a remote multilog.

[3] For simplicity we assume discrete time and use a global time notation. The theory does not assume that a site can observe the global time.

A specific protocol may have additional transition rules. As an example, let us encode a linearisable protocol [9], i.e., one in which an action takes effect at some instant in time, and actions execute in taking-effect order. We translate this to the following transition rule: "Only one action may be submitted per unit of time; if α is submitted at time t, then for any action $\beta \neq \alpha$: if $\beta \in \bigcup_j K_j(t-1)$ then $\beta \rightarrow \alpha$, otherwise $\alpha \rightarrow \beta$."

A replicated system based on pessimistic concurrency control, or *pessimistic system*, has transition rules that ensure that at every site and every time $S_i(t)$ is a prefix of $S_i(t+1)$. Otherwise the system is said *optimistic*.

3.2 Liveness conditions

While different replication algorithms maintain different consistency invariants, all of them must satisfy some liveness conditions for convergence. We identify two liveness conditions, one for the propagation protocol that distributes actions and constraints, the other for the decision algorithm that stabilises actions and multilogs.

The propagation protocol must ensure that all actions and constraints submitted to the system eventually reach all nodes.

Property 1 (Eventual Propagation) *A replicated system has the Eventual Propagation (EP) property iff every submitted action and constraint is eventually known everywhere:*

- $\alpha \in K_i(t) \Rightarrow \forall j : \exists t' : \alpha \in K_j(t')$
- $\alpha \rhd_{i,(t)} \beta \Rightarrow \forall j : \exists t' : \alpha \rhd_{j,(t')} \beta$
- $\alpha \rightarrow_{i,(t)} \beta \Rightarrow \forall j : \exists t' : \alpha \rightarrow_{j,(t')} \beta$

The decision algorithm must ensure that all locally known actions are eventually decided:

Property 2 (Eventual Decision) *A replicated system has the Eventual Decision (ED) property iff every submitted action is eventually decided:* $\alpha \in K_i(t) \Rightarrow \exists t' : \alpha \in Decided(M_i(t'))$.

ED implies that every action eventually becomes stable [16]. ED does not preclude the trivial implementation that makes every action dead; our framework does not rule this out, since it is a valid strategy if actions fail.

3.3 Mergeability and Uniform Local Soundness

We now discuss different definitions of consistency in our framework. The first one, Mergeability, captures the intuition that sites must not make conflicting decisions: a hypothetical omniscient observer would not see anything wrong. Mergeability generalises the classical *serialisability* property [1].

Property 3 *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 is not easy to ensure in a distributed setting. For instance, consider Site 1 has multilog $(\{\alpha\}, \emptyset, \{\text{INIT} \rhd \alpha\})$ and Site 2 has multilog $(\{\alpha\}, \{\alpha \rightarrow \alpha\}, \emptyset)$. They are both sound but not mergeable, as their union $(\{\alpha\}, \{\alpha \rightarrow \alpha\}, \{\text{INIT} \rhd \alpha\})$ is not sound.

Mergeability suggests that for safety, it is enough if all sites agree upon a deterministic decision strategy. For instance, a simple timestamp-based protocol can guarantee mergeability by ensuring that all sites order actions uniformly, using a global timestamp.

Under the EP liveness assumption, every submitted action and constraint is eventually received everywhere, so in effect every site becomes an omniscient observer. Then Mergeability reduces to the simpler Uniform Local Soundness (ULS) invariant that site multilogs are sound at all times: $\forall i, t : \Sigma(M_i(t)) \neq \emptyset$.

3.4 Eventual consistency

A classical consistency property for optimistic replication systems [15] is Eventual Consistency. It has been used to argue informally about the correctness of Grapevine [2] or Bayou [21].

Property 4 *A system is Eventually Consistent if, if every client stops submitting, and submitted actions are decided, then eventually every site will execute the same schedule, up to equivalence, and hence have the*

same final value:

$$\exists T : \forall i, t > T \Rightarrow \textit{No actions are submitted at } i$$
$$\Longrightarrow$$
$$\exists T', \forall t', t'', i, j : t' > T' \wedge t'' > T'$$
$$\wedge\, S_i(t') \in \Sigma(M_i(t')) \wedge S_j(t'') \in \Sigma(M_j(t''))$$
$$\Rightarrow S_i(t') \equiv S_j(t'')$$

Although eventual consistency simply captures the notion of replica convergence, it says little about the safety invariants satisfied by the algorithm before the system stabilises; these properties are captured by mergeability.

3.5 Common Monotonic Strong Prefix (CMSP)

Lamport's replicated state machine approach [10] mandates that all site execute exactly the same schedule. Clearly such a system is consistent, but this does not work for optimistic protocols where $S_i(t)$ is not necessarily a prefix of $S_i(t+1)$. However, even in an optimistic system, over time some actions will stabilise and form a prefix of all schedules. Such a system is consistent if the stable prefixes of different sites are equivalent. The system makes progress if the prefix grows.

Formally, a schedule P is a prefix of schedule S, written $P \ll S$, if $S \equiv S'$ where S' is a schedule of the form $P.Q$ for some sequence of actions Q.

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

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

We show that the actions in a CMSP are stable, and that the set of stable actions forms a CMSP [16].

3.6 Summary

We have presented four definitions of consistency, along with two liveness conditions. An interesting result is that under uniform assumptions, these definitions of consistency are equivalent. This may come as a surprise, since the operational definitions appear so different. In particular, under the eventual propagation and eventual decision liveness conditions, uniform local soundness (and hence mergeability) guarantees eventual consistency and the common monotonic strong prefix property. We provide a formal proof in our technical report [17].

4 Replication Systems and Decision Strategies

Consistency requires an agreement between all sites, which in the general case entails a consensus. For instance, mergeability forbids different sites from making conflicting decisions, thus requiring a consensus between deciding sites. Yet some practical protocols manage without this complexity, primarily by making assumptions about the distribution of constraints across actions. Here, we survey a few replication algorithms and their decision strategies.

Timestamped Replication In timestamped replication, all actions are assumed to have a unique timestamp. This timestamp induces a total $\rightarrow$ order on all actions even before they are submitted to the system. No decisions need to be made since all actions are guaranteed and ordered, hence stable, when submitted. Hence mergeability is guaranteed by default. The replication algorithm thus reduces to a simple propagation protocol that must satisfy the EP property.

A variation on the timestamped replication algorithm is one that uses the "last-writer wins" decision strategy. It assumes that each action modifies a single variable and when two actions modify the same variable, the action with the later timestamp should be effectively executed last. So, when two actions are received out of order, either they commute, or the one received later is converted to a no-op (identity action). As we argued in Section 2.1, this strategy makes all actions effectively

commute, and trivially guarantees mergeability, while allowing sites to execute actions without delay.

ESDS The ESDS protocol [7] assumes that actions only have acyclic causal constraints between them, of the form $\alpha \rhd \beta, \beta \rightarrow \alpha$. All actions are guaranteed at submission, the only requirement is that their causal predecessors must execute first. ESDS again reduces to a propagation algorithm that must satisfy the EP property, while maintaining the invariant that whenever α is propagated to a site, all β such that $\alpha \rhd \beta$ have already been propagated. This enables each site to easily keep track of the actions it can safely execute. Since actions do not commute, ESDS requires a distributed agreement for serialisation. All the sites participate in computing a total order for actions that is consistent with the causal order.

Bayou Many systems centralise consensus at a primary site. Bayou [21] assumes that the shared data can be partitioned into independent databases, each with its own primary site. Actions on different partitions commute and are assumed to have no constraints between each other. Primaries make decisions for their own actions and order them. Hence, the replication system consists of a propagation protocol satisfying EP that ensures that all actions reach their primaries, a primary decision strategy that ensures ED, and a propagation protocol that distributes the primary decisions to all sites. By centralizing the decision-making for each partition, mergeability for actions and constraints on a single partition are ensured, and by disallowing constraints between partitions, all site multilogs are mergeable.

Sufficient conditions for local decision If the constraint graph has some well-behaved properties, some decisions can be safely decentralised; in Section 5 we will derive an efficient decision protocol from the following observations. Consider for instance an action α that is involved in a single constraint $\alpha \rhd \beta$: then it is always safe to make α dead, regardless of the decision for β. Conversely, if α is only involved in $\gamma \rhd \alpha$, it is always safe to make α guaranteed, regardless of γ. This can be generalised to any acyclic $\rhd$ graph. Taking the example of a chain $\alpha_1 \rhd \ldots \rhd \alpha_n$ it is safe to either: make α_1 dead, then move on to α_2, left to right; or make α_n guaranteed, then move on to α_{n-1}, right to left. Since users don't like to see their actions aborted, guaranteeing in the right-left direction is preferable.

The decision regarding each α_i must consider $\rightarrow$ constraints. If α_i is not part of a $\rightarrow$ cycle, the decision may be either guarantee or make dead (although guaranteeing is preferable). If it is part of a $\rightarrow$ cycle, and all other actions in the cycle are guaranteed, the only sound decision is to make α_i dead; otherwise either decision is allowed.

Such local decisions may be sub-optimal. To ensure optimality, viz., that the smallest possible number of actions is made dead, it is necessary to consider the whole graph as in IceCube [12].

5 A decentralised replication algorithm

Consider a travel booking system, where airlines and hotels each manage their own primaries, but a user wants his hotel and flight bookings to happen atomically (all-or-nothing). Previous systems do not support this scenario: for instance Bayou imposes that all actions in a transaction have the same primary. We present a new algorithm, derived from the safe decision conditions from Section 4, that works for arbitrary constraint graphs, hence does not suffer this restriction.

5.1 Input assumptions

We assume the invariant that when α is in K_i, all constraints such that $\alpha \rhd \beta$ and $\beta \rightarrow \alpha$ (for any β) are known at i. Each action in A is eventually submitted at some site. In addition, each action α is assigned a unique primary site, $P(\alpha)$. We assume that two actions commute if and only if they have different primaries. Conflicting (mutually-excluding) actions are represented by $\rightarrow$ cycles. We assume the existence of a function $victim(c)$ that deterministically chooses one action from a subset of actions C.

Note that Bayou relies on the independence of primaries to enable distributed decision making. Each primary waits for actions and makes them guaranteed or dead without coordinating with other primaries. In contrast, our algorithm must consider $\rhd$ and $\rightarrow$ con-

straints between actions on different primaries.

5.2 Propagation module

We re-use the standard Bayou anti-entropy algorithm for propagating actions and constraints to all sites. The algorithm satisfies the eventual propagation property: every action and constraint submitted at some site eventually reaches all other sites. In addition, it maintains the invariant from the previous section.

5.3 The decision algorithm

Every primary must know for every action whether it is guaranteed or dead, and its execution order with respect to other non-commuting actions. To represent these decisions, each site maintains a set G_i of guaranteed actions, a set D_i of dead actions, and a relation $O_i \subseteq G_i \times G_i$ that totally orders all actions belonging to the same primary. The normal propagation module reliably distributes decisions among all sites.

Given these sets, the schedule executed at a site is any schedule that contains all guaranteed actions, no dead actions, and obeys the MustHave constraints in $\rhd_i$ and the ordering constraints in $\rightarrow_i$ and O_i. For uniform local soundness, such a schedule must always exist. For eventual decision to hold, all actions in K_i must eventually be included in G_i or D_i.

The decision algorithm runs concurrently with the propagation module. An action is first submitted to the system, then it becomes ready for a decision, it may become guaranteeable, and finally it becomes guaranteed or dead. We present the decision algorithm in terms of these states.

Ready Actions An action α is said to be *ready* at its primary $P(\alpha)$ if

- All β such that $\alpha \rhd \beta$ are known at $P(\alpha)$
- All β such that $\beta \rightarrow \ldots \rightarrow \alpha$ are known at $P(\alpha)$.

Each of these conditions imposes a wait before any decision on α can be taken. A primary has a set of ready actions from which it chooses the next action to make a decision on.

Guaranteeable Actions Once all the constraints on an action α are collected, the primary begins the process of discovering whether α can be guaranteed. In particular, since it knows the closure of Before and MustHave relations, it can detect all the decision cycles between actions. For an action to be guaranteeable, all the actions it MustHave should be guaranteeable and at least one member of each $\rightarrow$ cycle it belongs to should be dead. This stage comprises the following steps:

- Compute the set M of all actions in a $\rhd$ cycle with α. Let the set of remaining actions it MustHave be designated M'.
- Compute the set C of action sets representing cycles of $\rightarrow$ involving α.
- Wait for all the actions in M' to become guaranteed. If any of these actions becomes dead, α is now known to be dead; exit.
- For each cycle c in C, designate $victim(c)$ to be dead. If this action is α, exit.
- Designate α as guaranteeable.[4]
- Send messages to all primaries with actions in M saying that α is guaranteeable.

We again rely on the propagation module to distribute the guaranteeable actions to related primaries.

Guaranteed actions In the case of $\rhd$ cycles, all members of the cycle must agree to either be guaranteed or be dead. The final steps before guaranteeing are as follows:

- Wait until either some action in M is dead, or all actions in M are guaranteeable. If the former, α is now known to be dead; exit.
- Wait until all β such that $\beta \rightarrow \ldots \rightarrow \alpha$ and $P(\beta) = P(\alpha)$ have been decided.
- Guarantee α and order it after all guaranteed β with $P(\beta) = P(\alpha)$

Dead actions In the process of computing guaranteeable and guaranteed actions, we identify two conditions in which an action becomes dead: either when one of the actions it MustHave is dead (either down a $\rhd$ chain,

[4] Some systems may elect to make α dead at this point according to their own strategies. For instance, Bayou checks a predicate, called the "dependency check," attached to each action.

or in a $\triangleright$ cycle), or when it is designated as the victim in a $\rightarrow$ cycle.

The choice of action to make dead in a $\rightarrow$ cycle can be arbitrary. In general it is safe to make one or more actions in such a cycle dead, as long as this is propagated up any $\triangleright$ chain. However, making too many actions dead, or choosing the wrong action to make dead, can have a negative impact on performance.

Summary of decision algorithm We now summarize the steps for deciding action α. Assume α was submitted at site j.

1. Through epidemic (or other) communication, α is eventually known at its primary site i, $P(\alpha) = i$.
2. The propagation module at site i communicates with other sites, discovering all β such that: $\alpha \triangleright \ldots \triangleright \beta \vee \beta \rightarrow \ldots \rightarrow \alpha$. The action becomes ready.
3. For each cycle c of $\rightarrow$ involving α, if $victim(c) = \alpha$, then decide α is dead (e.g., add constraint $\alpha \rightarrow \alpha$) and exit.
4. Partition all β such that $\alpha \triangleright \beta$, into subsets M and M', according to the following property: actions in M are such that $\beta \triangleright \alpha$, those in M' are not.
5. Wait until: either some action in M' is known to be dead; or all actions in M' are known to be guaranteed. In the former case, α is now known to be dead; exit. In the latter, α is now guaranteeable.
6. To all actions in M, send a message saying that is α is guaranteeable.
7. Wait for either some action in M to be known to be dead, or for all actions in M to be guaranteeable. In the former case, α is now known to be dead; exit. In the latter, decide α is guaranteed (e.g., add INIT $\triangleright \alpha$).
8. The final execution order of α is given by its $\rightarrow$ relations. Wait for all β such that $\beta \rightarrow \ldots \rightarrow \alpha \wedge P(\alpha) = P(\beta)$. Execute α after all such actions that are guaranteed.

5.4 Correctness

To prove the consistency and convergence of the algorithm, we rely on eventual propagation, on eventual decision, and on uniform local soundness.

The propagation module is fashioned on standard anti-entropy protocols and reliably delivers all actions, constraints, and decisions. To prove that the decision algorithm eventually decides every action, we show that all the wait conditions in the algorithm are eventually satisfied, i.e., there are no wait-for cycles. For uniform local soundness, we argue that every decision extends the set of constraints in a sound manner by performing a step-by-step case analysis on the algorithm.

5.5 Extensions for partial replication

Up to now we assumed that all data is replicated at every site. Let us now consider partial replication: shared data is partitioned into n disjoint *databases* $D^1, \ldots, D^n$, and we allow a site to replicate an arbitrary subset of the databases (as long as every database is present on at least one site). Actions are correspondingly partitioned into subsets $A^1, \ldots, A^n$. A site replicating D^i should receive submitted actions that are in A^i, and the constraints that involve such actions. It does not need to receive actions or constraints for databases it does not replicate.

Both our correctness conditions and our distributed algorithm extend naturally to partial replication with constraints across partitioned data. The analysis and presentation of this modified algorithm is left to a future paper; here we sketch some details.

The $\triangleright$ constraint 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 partitions, Split MustHave, noted $\trianglerighteq\!\!\!\triangleright \subseteq A \times A$. The definitions of mergeability and eventual consistency can then be extended in terms of this new $\triangleright$ operator.

The distributed algorithm stated above uses full replication only in computing the closure (and cycles) of $\triangleright$ and $\rightarrow$. Under partial replication, this computation must be done in a distributed manner. We adopt Manivannan and Singhal's distributed knot detection algorithm [11] for this purpose.

6 Related work

IceCube is a general-purpose system supporting optimistic replication and cooperative work [12], based on actions and constraints. Experience with IceCube shows that relatively complex applications can be readily encoded in this framework. Its decision algorithm is centralised and computes an optimal schedule given an arbitrary graph of actions and constraints. Although the problem is NP-hard, IceCube uses efficient heuristics and manages to execute in almost linear time in the common case.

Our survey of optimistic replication [15] motivated us to understand the commonalities and differences between protocols.

Chong and Hamadi [3, 17] proposed a decentralised decision algorithm based on constraint satisfaction principles, which inspired our algorithm in Section 5.

The relations between consistency and ordering have been well studied in the context the causal dependence relation [13, 14]. Our simpler and modular primitives clarify and generalise this analysis. The primitives are common to all protocols, as are the significant events of actions becoming guaranteed, dead, serialised, decided and stable.

Lamport's state-machine replication [10] broadcasts actions to all sites and ensures consistency because each site executes exactly the same schedule. Our CMSP property generalises this definition. Sousa et al. [19] generalise Lamport's state-machine approach to the commitment of partially replicated databases.

Much formal work on consistency focuses on serialisability [1, 5]. Mergeability constitutes a generalisation of serialisability.

The X-Ability theory [8] allows an action to appear several times in the same schedule if it is idempotent; for instance, retrying a failed action is allowed. Schedules are tested for equivalence after filtering out such duplicates. It would be interesting to encode their approach in our formalism, and analyse their assumptions, which are quite strong. This is left for future work.

Our approach has many similarities with the Acta framework [4, 5]. 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 analyse protocols at a 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. Acta takes serialisability as the definition of consistency, and does not deal with partial replication.

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

7 Conclusions and future work

We presented a formalism for describing replication protocols and consistency. Our significant subsets are common to the many replication protocols that can be described in our language. We generalise a number of classical formulations of the consistency property and prove them equivalent. This underscores the deep commonalities between protocols that appear quite different on the surface. Although consistency entails global consensus in the general case, we exhibited sufficient conditions for making local decisions. We derived a new distributed decision algorithm, which supports multiple primaries, constraints across primaries, and can be extended to handle partial replication. Our results apply to a broad range of protocols, both pessimistic and optimistic.

This paper only presented the intuitions; the interested reader will find a fully formal treatment in our technical report [16]. That report also contains a detailed description for a variety of diverse classical replication protocols, including consistency proofs.

The formalism rests upon only two binary constraints. This makes it easy to prove properties, and is powerful enough to incorporate all the classical replication protocols. However the semantics of some applications (e.g., a shared bank account) demand more powerful primitives. A possible direction is to generalise constraints to be n-ary and our significant subsets to patterns. Then the crucial safety property would be that the guaranteed and dead subsets are disjoint.

Acknowledgments

We thank Fabrice le Fessant for his participation to early stages of this work, Yek Chong and Youssef Hamadi for their contributions on a decentralised decision algorithm, and Tony Hoare, Miguel Castro and Patrick Valduriez for their encouragement and suggestions.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. 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. *Commun. ACM*, 25:260–274, Apr. 1982.

[3] Y. Chong and Y. Hamadi. Distributed IceCube. Private communication, Jan. 2004.

[4] P. K. Chrysanthis and K. 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] P. K. Chrysanthis and K. 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] F. 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] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(Special issue on Distributed Algorithms):113–156, 1999.

[8] S. Frølund and R. Guerraoui. X-Ability: A theory of replication. In *Symp. on Principles of Dist. Comp. (PODC 2000)*, Portland, Oregon, USA, July 2000. ACM SIGACT-SIGOPS.

[9] M. Herlihy and J. Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[11] D. Manivannan and M. Singhal. An efficient distributed algorithm for detection of knots and cycles in a distributed graph. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):961–972, October 2003.

[12] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, Catania, Sicily, Italy, Nov. 2003.

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

[14] M. Raynal and M. 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), Sept. 1993.

[15] Y. Saito and M. Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, Oct. 2003. ftp://ftp.research.microsoft.com/pub/tr/tr-2003-60.pdf.

[16] M. Shapiro and K. Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, Mar. 2004. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-14.pdf.

[17] M. Shapiro, K. Bhargavan, Y. Chong, and Y. Hamadi. A formalism for consistency and partial replication. Technical Report MSR-TR-2004-58, Microsoft Research, Cambridge, UK, June 2004. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-58.pdf.

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

[19] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *Int. Symp. on Network Comp. and App. (NCA'01)*, pages 298–309, Cambridge MA, USA, Oct. 2001. IEEE.

[20] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *Trans. on Comp.-Human Interaction*, 5(1):63–108, Mar. 1998. http://doi.acm.org/10.1145/274444.274447.

[21] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. 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), Dec. 1995. ACM SIGOPS. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[22] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Computer Supported Cooperative Work*, pages 171–180, Philadelphia, PA, USA, Dec. 2000.