The three dimensions of data consistency

Marc Shapiro, Nishith Krishna

The three dimensions of data consistency

Marc Shapiro*
INRIA, Rocquencourt, France and LIP6, Paris, France

Nishith Krishna
Courant Institute of Mathematical Sciences, New York University, New York, USA

Abstract

Replication and consistency are essential features of any distributed system and have been studied extensively, however a systematic comparison is lacking. Therefore, we developed the Action-Constraint Framework, which captures both the semantics of replicated data and the behaviour of a replication algorithm. It enables us to decompose the problem of ensuring consistency into three simpler, easily understandable sub-problems. As the sub-problems are largely orthogonal, sub-solutions can be mixed and matched. Our unified framework enables a systematic exploration of pessimistic vs. optimistic protocols, full vs. partial replication, strong vs. weak consistency, etc.

1 Introduction

Replicating shared data improves availability and performance but raises consistency issues. There are many different replication protocols which differ in subtle and confusing ways. Further complications come from interaction with object semantics, from using optimistic techniques and from partial replication (e.g., caching). It is difficult to understand the behaviour of these protocols, to compare their properties and to reason about their correctness.

To address this problem, we use the *Action-Constraint Framework* (ACF) [10]. This formalism enables us to describe in a unified way the semantics of shared data and applications, user intents, and replication protocols. It makes it possible to prove consistency properties, even in the presence of optimistic replication or partial replication.

The ACF represents the information in a replicated system as a (replicated) graph called a *multilog*. Its nodes are operations, or *actions*, connected by three kinds of edges, called *constraints*, which reify concurrency invariants that must be maintained. Maintaining consistency can be divided into three simple problems on sub-graphs of a multilog, Conflict Breaking, Agreement and Serialisation respectively. This divide-and-conquer approach clarifies the problem space, the spectrum of solutions, and the trade-offs. This analysis can serve either to understand existing protocols or as a guide for the design of new protocols

2 The Actions-Constraints Framework

2.1 Basic definitions

A distributed system contains is made of computers or *sites* $i, j, \ldots$, that replicate some (unspecified) data. A client at a site submits *actions* α, β, $\ldots$, operating on the shared data, related by *constraints* that reify semantic relations. Constraints are posed by users, data types, applications, or protocols.

Sites communicate by sending actions and constraints that they know. Messages are asynchronous but are eventually delivered reliably. For simplicity we assume an epidemic communication style, but this is not essential.

*Work performed while at Microsoft Research, Cambridge, UK

Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR), CNAM, Paris, France, 2–4 Nov. 2005

The collection of actions and constraints known at site i at time t is *multilog* $M_i(t) = (K_i, \rightarrow_i, \lhd_i, \nvdash_i)(t)$ (when there is no ambiguity we will abbreviate to just $M = (K, \rightarrow, \lhd, \nvdash)$), where K is the set of actions known so far, and the others are sets of *constraints*, explained shortly; all are non-shrinking over time.

Each site independently executes actions in K it knows of so far in some serial *schedule* $S_i(t)$ (abbreviated S). A schedule must be *sound*, meaning that: it starts with the special action INIT (the initial state); every action in K appears in S zero or once; and it obeys the constraints:

- "α Before β:" $\alpha, \beta \in S \wedge \alpha \rightarrow \beta \Rightarrow \alpha <_S \beta$
- "β MustHave α:" $\beta \in S \wedge \alpha \lhd \beta \Rightarrow \alpha \in S$

The set of sound schedules at site i at time t is $\Sigma(M_i(t))$ (abbreviated $\Sigma(M)$).

Two actions commute if executing them in either order results in an equivalent state (where state equivalence is defined by the application); otherwise they are *non-commuting*, noted $\alpha \nvdash \beta$. Two schedules are equivalent if they differ only by swapping adjacent commuting actions.

By combining these three constraints in different ways we are able to capture the concurrency semantics of a number of interesting applications, e.g., shared calendars [8, 7], a replicated file system [11] or a shared document editor [6]. We do not claim that this particular choice of constraint language is by any means unique or complete, but it is sufficiently expressive for the applications and protocols that we looked at. As we shall see later, it enables a decomposition of consistency protocols into relatively independent subproblems.

ACF can model non-deterministic protocols and optimistic protocols. Of course, ACF can also model a pessimistic system: this is where $\forall \alpha : \alpha \lhd$ INIT, hence every known action must execute and cannot roll back. Similarly, ACF can model a deterministic system: it is one where the constraints ensure that $|\Sigma(M)| = 1$ (modulo schedule equivalence) always.

2.2 Significant subsets

Any useful system must eventually make irrevocable scheduling decisions. *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 $\beta \lhd \alpha$ then $\beta \in Guar(M)$. For space reasons we omit the formal definition of the following subsets, but they do not pose any complication. *Dead* actions $Dead(M)$ do not execute in any schedule. An action is *serialised* ($Serialised(M)$) if it is ordered with respect to all non-commuting actions that execute. An action is *decided* once it is either dead, or both guaranteed and serialised: $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$.

To make some action α guaranteed, it suffices to add a constraint such as $\alpha \lhd$ INIT. Similarly, adding $\beta \rightarrow \beta$ suffices to make action β dead.

2.3 Consistency

Each site has its own view or multilog and (conceptually) executes schedules independently. A system is *consistent* iff it satisfies both

- (liveness) *Eventual Decision*: every action is eventually decided at all sites; and
- (safety) *Mergeability*: the union M of any combination of local multilogs viewed at any time is such that $Guar(M) \cap Dead(M) \neq \emptyset$.

Mergeability is a generalisation of the well-known serialisability condition. We have proved elsewhere [9] that it is equivalent to Eventual Consistency (i.e., if clients stop submitting new actions, all replicas eventual reach the same state) [12] in the presence of sufficiently strong liveness conditions.

3 The three dimensions of consistency

Consistent replication is ensured by any distributed algorithm that ensures mergeability and eventual decision. The difficulty is satisfying mergeability even when different sites run instances of this algorithm asynchronously and in parallel, on differing views. We propose a decomposition of the consistency problem into simpler sub-problems. For each one we propose an abstract sub-algorithm with the right properties, for which there are different concrete implementations. The output sub-algorithm composes the results of the others.

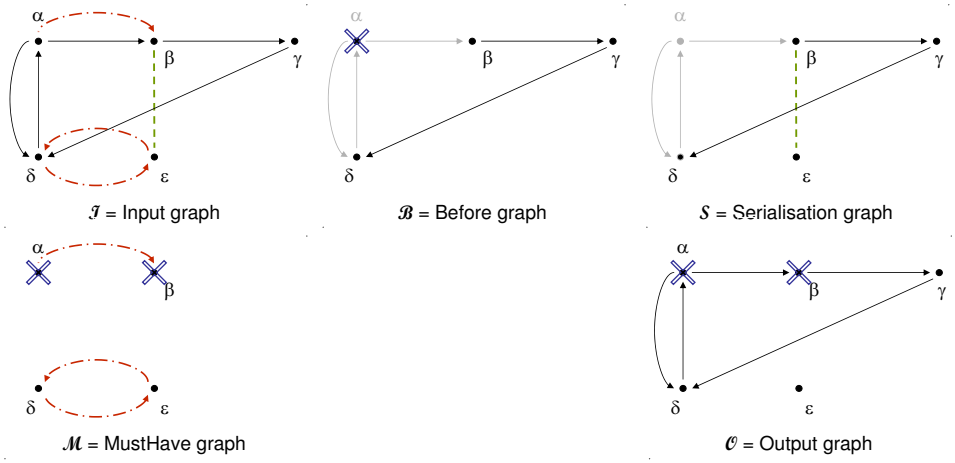

Figure 1: Example graph and decomposition. Note causal dependence $\alpha \rightarrow \beta \wedge \alpha \lhd \beta$, conflict $\alpha \rightarrow \delta \wedge \delta \rightarrow \alpha$, and atomicity $\delta \lhd \epsilon \wedge \epsilon \lhd \delta$.

3.1 Graphs

Figure 1 shows an example input graph $\mathcal{I}$, the corresponding derived $\mathcal{B}$, $\mathcal{S}$ and $\mathcal{M}$ graphs at different stages, and the output graph $\mathcal{O}$. The nodes of the $\mathcal{I}$ graph represent the actions; edges are of three types, $\rightarrow$ (solid arrows in the figure), $\alpha \lhd \beta$ (mixed-dashed arrows) and $\nmid$ (short dashes) edges. The $\mathcal{B}$, $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{O}$ graphs are initialised from the $\mathcal{I}$ graph, but evolve as described hereafter.

Each action has a colour, either grey, black or white. When output terminates, black nodes correspond to dead actions, and white nodes to guaranteed actions. A black (resp. white) action remains black (resp. white) thereafter; if a node is blackened in any of the graphs, it eventually appears black in all of $\mathcal{B}$, $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{O}$.[1] Colour changes and new edges created by serialisation are transmitted between graphs (and between replicas) by asynchronous but reliable messages.

3.2 Conflict breaking

By eventual decision, every action must become either dead or guaranteed. However, for mergeability, not all actions in a $\rightarrow$ cycle can be guaranteed. *Conflict breaking* ensures that in any $\rightarrow$ cycle at least one action is dead.

Abstract conflict-breaking algorithm *Initialise the $\mathcal{B}$ graph with the Before edges and the nodes connected by such edges in the $\mathcal{I}$ graph. Repeatedly, do one of the following: • Choose some grey node; mark it black; • Delete some black node and its edges from the graph. Terminate when the graph is acyclic.* ■

Caveat: some serialisation algorithms create new cycles.

There is a range of concrete implementations that satisfy this specification, with different cost-quality trade-offs.[2] As long as at least one node in every cycle is blackened, the choice can be somewhat arbitrary. Some choices are better than others; for instance in Figure 1, blackening α breaks two cycles, whereas blackening β would only break one, and ϵ none.

One possible implementation, which we call B-TotalOrder orders nodes by timestamps and blackens α whenever $(\alpha \rightarrow \beta \wedge \mathsf{id}(\alpha) > \mathsf{id}(\beta))$. Alternatively, B-HighDegree blackens the node with highest degree in some cycle, and B-IceCube uses an optimisation algorithm to minimise the number of nodes to blacken [4].

[1] To simplify the exposition, we will assume the input graph does not change during execution of the protocol. Initially all nodes are grey; at the end they are all white or black. Thereafter, the application may add new grey nodes, and the protocol execute again, its nodes initially a mixture of white, grey and black.

[2] An algorithm is of higher quality than another if the former blackens fewer actions than the latter.

3.3 Agreement

Agreement algorithm *Initialise the $\mathcal{M}$ graph with the MustHave edges and the nodes connected by such edges in the $\mathcal{I}$ graph. When some node is black, blacken all its successors. Repeat; terminate when conflict breaking has terminated and there are no more black marks to propagate.* ■

Whenever conflict breaking blackens a node in the $\mathcal{B}$ graph, agreement propagates the black mark in the $\mathcal{M}$ graph. Therefore, agreement cannot terminate until conflict breaking has terminated.

3.4 Serialisation

Eventual decision requires that an execution order be chosen for every pair of non-commuting actions that are not dead. This is the job of the serialisation algorithm. It operates on the $\mathcal{S}$ graph, which contains both (directed) Before edges and (undirected) NonCommuting edges.

Abstract serialisation algorithm *Initialise the $\mathcal{S}$ graph from the $\mathcal{I}$ graph, copying directed $\rightarrow$ edges and undirected $\nleftrightarrow$ edges, and the nodes connected by such edges. Repeatedly, do one of the following: • Choose a black node; delete that node and any associated edges. • Choose some undirected edge and insert a directed edge, in some direction, between its nodes; add the new edge also to both the $\mathcal{B}$ and $\mathcal{O}$ graphs. • Choose two nodes connected by both a directed and an undirected edge; delete the undirected edge. Terminate when the $\mathcal{S}$ graph contains no more undirected edges.* ■

Concrete implementations include S-Conservative, a standard in databases, which replaces an undirected edge with a pair of directed edges in both directions. This new cycle will be broken by conflict breaking. S-TotalOrder chooses the direction of an inserted edge deterministically using totally ordered unique identifiers [2].

Serialisation might add new cycles in the $\mathcal{B}$ graph; therefore in the general case, conflict breaking must not terminate until serialisation has terminated. There is a difficult trade-off. New cycles causes more dead actions than strictly necessary. We have designed a serialisation algorithm that completely avoids this, at the cost of added synchronisation. Space restrictions preclude a full description.

3.5 Output

The output graph $\mathcal{O}$ is a scheduling graph. Its nodes are the same as those of the input, and are either black or white. Its white nodes are partially ordered by Before edges, combining those that were in the original input and those added by serialisation.

Output algorithm *Initialise $\mathcal{O}$ with all nodes, and with the directed Before edges of the $\mathcal{I}$ graph. When serialisation adds a directed edge, also add it to $\mathcal{O}$. When conflict breaking, agreement and serialisation are all terminated, make every black node α dead, by adding $\alpha \rightarrow \alpha$ to the multilog M. Whiten every remaining grey node β and make it guaranteed by adding $\beta \lhd$ INIT. Terminate.* ■

4 Discussion and conclusion

Lamport's happens-before relation [5] connects events in a distributed system. We distinguish happens-before from semantic causality, which we note $\alpha \rightarrow \beta \wedge \alpha \lhd \beta$. Separating the constraints $\rightarrow$ and $\lhd$ make it easier to decompose systems into basic components. Our primitives are similar to Acta [1], a formalism for describing transaction systems. Whereas Acta wires-in database concepts such as transactions and focuses on advanced transaction models, our goal has been to understand replication from first principles, and as a result we consider a much finer granularity. There are some similarities between our formalism and X-Ability [3], a theory of replication in the presence of faults, in reactive model. Converging the two theories is an area of future research.

The ACF provides a simple, formal language for reasoning about replicated data systems. Mergeability and eventual decision constitute universal correctness conditions for consistency. Our framework enables us to break protocols down into easily-understood sub-problems, solved by abstract sub-algorithms, each of which admits a number of concrete implementations. This exposes three dimensions over which replication algorithms vary: conflict breaking, serialisation, and

output. (There is not much variation in the fourth sub-problem, agreement.) This approach makes it easier to understand and compare existing protocols and to design superior new protocols. Although lack of space precludes justification, we claim that the algorithms remain correct even when multiple instances execute in parallel and asynchronously, or in the presence of partial replication. Safety is maintained in the presence of non-byzantine failures. The algorithms extend naturally to the case where the action-constraint graph changes dynamically.

Modelling the consistency properties of a distributed system as a constraint graph provides an effective explanatory tool for understanding and designing consistency algorithms. It enables our decomposition of consistency into sub-problems that can be solved relatively independently. We make no claim that this decomposition is necessary or superior to any other approach. Indeed, it is clearly sub-optimal (in terms of quality), since each sub-algorithm operates on partial information only; we only claim that it simplifies reasoning. Previously [8] we designed an optimal scheduler; it was not decomposed (full information being necessary for optimality) and was centralised in order to avoid a consensus.

We plan to continue this work to gain a fuller understanding of replication. There is an implicit connection between the kind of constraints posed by applications and the consistency protocol used; more work is needed to formalise this connection. In this paper we focused mainly on correctness; future work is needed to explore the real-world performance trade-offs. We believe this is greatly eased by our decompositional approach.

References

[1] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proc. Int. Conf. on Mgt. of Data*, pages 194–203, Atlantic City, NJ, USA, May 1990.

[2] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113–156, 1999.

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

[4] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *20th Symp. on Principles of Dist. Comp. (PODC)*, Newport, RI, USA, August 2001.

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

[6] James O'Brien and Marc Shapiro. An application agnostic replication system for ubiquitous computing. Technical Report MSR-TR-2004-64, Microsoft Research Cambridge, July 2004.

[7] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.

[8] 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.

[9] Marc Shapiro and Karthik Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, March 2004.

[10] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, Grenoble, France, December 2004.

[11] 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.

[12] 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.