The Distributed Computing Column

BY

Stefan Schmid

Aalborg University
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark

This issue of the distributed computing column includes two articles:

1. Ittai Abraham and Dahlia Malkhi present an interesting perspective on Blockchain consensus protocols, through the lens of distributed computing, and discuss the relationship to Byzantine Fault Tolerant (BFT) protocols.

2. Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira present a survey of the mathematical structures and compositional properties of state-based replicated data types that support the implementation of eventually consistent, geo-replicated data management solutions.

Enjoy!

The Blockchain Consensus Layer and BFT

Ittai Abraham Dahlia Malkhi
Vmware Research Group
VMware
{iabraham, dmalkhi}@vmware.com

Abstract

In this paper, we analyze Blockchain consensus protocols in the lens of the foundations of distributed computing. Our goal is to present analogies and connections between Blockchain protocols and Byzantine fault tolerant (BFT) protocols. We also discuss opportunities to consider hybrid solutions.

Keywords: Blockchain, Byzantine Agreement, BFT

1 Introduction

In the early 2000's, a group of activists advocating the wide-spread use of cryptography and privacy-enhancing technologies were engaging over the *cypherpunks* mailing-list in an effort to create an anonymous, monitor-free digital cash. Step by step, they jointly built the ingredients that eventually lead to the emergence of Bitcoin in 2009. In recent years more crypto-currency variants have emerged. As of late 2017, the market cap associated with Bitcoin is over $70 billion and the total crypto-currency market cap is about twice that.

The steps leading to the construction of Bitcoin harness deep ideas and methods from published academic works. Its incredible market cap reflects the public trust in the robustness and soundness of the technology, without any company or institution backing it.

At the core of Bitcoin is a method for reaching agreement on a shared chain of blocks where each block contains a sequence of transactions. This core is called the Blockchain. In many ways the Blockchain is the most intriguing and innovative aspect of Bitcoin. In this paper, we study Blockchain through the lens of the theory of distributed computing.

To put this exploration in context, let's take a quick perspective on the full Bitcoin approach, and explain what challenge Blockchain addresses. Bitcoin relies heavily on the idea of *computational puzzles* as proof-of-work (PoW). The idea of using computational puzzles first appeared in the pioneering work of Dwork and Naor in [17]. Similar approaches were taken to fight email spam by forcing senders to work by Hashcash [8]. Aspnes et al. [7] suggested to use computational puzzles for preventing Sybil attacks in a Byzantine setting.

The cypherpunks were interested in a much more ambitious use-case of crypto-puzzles such as Hashcash, the use of computation power as means for minting crypto-currency. They recognized that PoW provided scarcity and uniqueness, two necessary ingredients for creating value. However, the main challenge was to provide users with a decentralized, anonymity-preserving solution to protect against double-spending. At first, this seems like a daunting problem: How can you track spendings and preserve privacy at the same time?

A couple of ideas that eventually led to the Blockchain solution were posted to the cypherpunks mailing list. Dai proposed b-money [12], a crypto-currency system that already uses crypto-puzzles for minting digital currency, in which participants themselves track all digital account balances. This was not a very practical approach because it relied on a timely multicast channel in a peer-to-peer setting, and rather than preventing double spending, involved rather cumbersome remediation mechanisms. Szabo enhanced this idea with the Bit Gold crypto-currency [38]. The Bit Gold system uses Haber and Stornetta's hash-chaining [22] to create a secure ordering of transfers, a transaction ledger. The position of a transaction in the ledger determines the latest owner of a digital asset. The question was who is in charge of maintaining the ledger. Bit Gold used Byzantine quorums [31] to maintain the ledger replicated among all users. Since there was no way to validate membership in Bit Gold, quorums were not enough to provide consistency guarantees.

The challenge of preventing double-spending in a permissionless setting thus remained unsolved until the introduction of the Bitcoin Blockchain protocol. The last piece of a full crypto-currency solution was finally addressed.

Blockchain uses PoW as a way to obtain several goals at once. It is a way to mint currency but more importantly it is a key ingredient in the Bitcoin Blockchain protocol that reaches agreement and prevents double spending. Newly minted blocks are spread among the miners over a peer-to-peer network, and each miner keeps the longest chain as the winning chain, even if it means overturning earlier segments of the chain. Since winning the longest chain means solving cryptographic puzzles in each block, overturning a long tail segment is hard. For this reason, blocks "buried" deep in a miner's chain are typically considered committed with high probability.

1.1 A Layered view

The Blochain technology is more than a consensus engine. Zooming out a bit, we advocate a *layered view*, which decomposes Blockchain into different components that each deals with separate concerns.

Layer-1: State-machine replication (SMR). The bottom layer is focused on an infrastructure for storing an immutable sequence of transaction-blocks among distrustful parties.

From a foundational standpoint, this layer builds a chain of *consensus* blocks, a problem that has received tremendous attention in the distributed systems arena. As evidenced from the above brief history, Bitcoin incorporates several deep ideas that have been around for quite a while, some more than two decades. Yet the consensus engine, which achieves agreement among distrusting parties in a scalable settings with unknown participants, seems very different than the classical methods for Byzantine fault tolerance (BFT).

This writeup is dedicated to an exploration of the analogous replication problem model with Byzantine fault tolerance in the distributed computing literature and relating BFT to Blockchain.

Layer-2: Smart contracts. The middle layer takes a simple shared data structure and exposes rich, high level, business relevant abstractions. This allows users and applications to use advanced cryptography tools to generate *smart contracts* [37]. Contracts allow to programatically facilitate, verify, or enforce the negotiation or performance of a business transactions.

They are embedded as Blockchain SMR commands, hence all participants in share the responsibility of executing them, their logic is immutable, and their audit-trace is open. A future writeup will overview the foundations of this layer.

Layer-3: Services. Finally, the top layer, *applications*, is where the customer value is created. There is great excitement about the possibilities that shared provenance of asset and identity tracking brings. Not surprisingly, many of them are in the financial arena, but not only, and it is yet to be seen what are the "killer-apps" for Blockchains.

1.2 Roadmap

In this paper, we analyze Blockchain consensus protocols in the lens of the foundations of distributed computing. In §2, we present the algorithmic foundation of Nakamoto Consensus (NC), explaining how it solves (with high probability) the state-machine replication (SMR) problem. In §3, we attempt to relate NC to the classical literature on Byzantine fault tolerant SMR (BFT). We continue in §4 to overview several approaches for bringing the two paradigms together. We conclude in §5.

2 Nakamoto Consensus through the lens of the theory of distributed computing

Distributed protocols often have some desired properties they wish to obtain. For example, Byzantine agreement protocols aim to solve the "consensus" problem and it is often said that Blockchains solve the "double spending" problem. It is crucial to understand under what conditions do these protocols obtain their desired properties. In distributed computing theory we typically start by defining an adversary model and then prove that the desired properties hold in any execution against such an adversary.

2.1 Adversary Model

In the traditional distributed computing literature, the most common adversary assumption is the *Threshold Adversary Model*. In this model there is typically an assumption of a threshold gap between two parameters:

1. The total number of parties, often denoted by n.

2. The total number of parties the adversary can control, often denoted by f.

The typical *threshold* chosen is either of a minority $n > 2f$ or a third $n > 3f$. This model is sometimes also called the "permissioned model" to indicate that it is not the case that anyone can join the group of n parties, but rather one needs to have a "permission to join". Indeed in this model we tyically assume there is a fixed set n participants.

One of the major model modifications that Bitcoin considers is the alternative *Computational Threshold Adversary (CTA)* model. In this model, instead of bounding the total number of parties the adversary controls relative to the total number of parties, the model bounds the

total amount of computational power the adversary has relative to the total available computational power of all parties. To push the analogy we can formally define:

1. The total amount of computational power, denoted by N_C.

2. The total amount of computational power the adversary can control, denoted by F_C.

The assumption in Bitcoin is that the adversary controls a minority of computational power. We can denote this by a *threshold* of $N_c > 2F_C$. In some cases it helps to assume the adversary controls some ϵ fraction less than a majority, $N_C > 2(1 + \epsilon)F_C$.

We note that the formal definition of "computational power" needs a concrete materialization. For example, in Bitcoin, this assumption builds on mechanisms that harnesses the difficulty of inverting SHA1.

This model is often also called the "permissionless model" to indicate that there is no explicit membership protocol, anyone who can solve cryptographic puzzles can join the system.

A new model modification is being debated and considered lately. The idea is to bound the adversary by allowing him a minority *stake* in some abstract finite resource. Indeed, instead of bounding the relative number of parties or the relative computational power of the adversary, one can imagine bounding some other limited resource. In a crypto-currency use case it is very natural to use the crypto-currency itself as the limited resource.

This leads to the *Stake Threshold Adversary (STA)* model. In this model, there is some finite abstract resource we will call R. Again we can define:

1. The total amount of resource R, denoted by N_R.

2. The total amount of resource R that the adversary can control, denoted by F_R.

For example, the underlining assumption in Ethereum's Proof-of-Stake approach can be modeled as an adversary that controls a third of the resource R (in their case of the Ethereum crypto-currency). We can simply denote this as a *threshold* assumption $N_R > 3F_R$.

In many ways this third model is a generalization of the previous two. The classic threshold model is just a one-party one-vote resource and the computational threshold model just using computation as the limited resource.

We note that in all three models there is a possibility to dynamically modify the parameters as long as the threshold remains the same. These dynamic changes often bring more challenges for protocol solutions.

An additional power of the Proof-of-Stake approach is that it allows for punishing parties (by "slashing" their stake) if they can be publicly detected as malicious. We briefly discuss this next.

2.2 Game theoretic model

Instead of designing protocols against a malicious adversary, an alternative approach is to design protocols that form a threshold coalition resilient equilibrium [3, 4]. In both models the potential deviating parties are limited by a threshold. The main difference is that in the malicious model the deviating parties can act completely arbitrarily while in the game theoretic model we often assume the deviating coalitions acts rationally while attempting to maximize their utility.

In the crypto-currency scenario it is often natural to define the utility simply as a function of the crypto-currency itself. The main mechanism used is the fear of punishment. This leads to the idea of each member putting some deposit (stake) in such a way that if a deviating member is publicly verified as malicious then its deposit can be deleted or reduced. This approach appears in Wei Dai's b-money [12] and in recent Casper suggestions [9].

In the rest of this section, we analyze Bitcoin in the Computational Threshold Adversary model, and leave the game theoretic model for a future survey.

2.3 The Computational Threshold Adversary model

We now discuss the Computational Threshold Adversary model in more detail. A key element of the CTA model is the notion of Proof-of-work (PoW): The adversary cannot prove more work than its threshold share of computation power.

Synchrony. We note that bounding the computational power is often meaningless as PoW in a fully asynchronous system. If network delay can be unbounded then the adversary can boost its computational power just by making the non-faulty parties incur much higher delays. It is therefore commonly accepted that the CTA model incorporates in it some degree of synchrony assumptions [20, 13, 35, 6]. A more detailed analysis of network delays appears in [34]. This is in contrast to the traditional threshold adversary model where both asynchrony and synchrony models can be considered.

PCNELE. Bitcoin uses PoW to implement a leader election "oracle" with several interesting properties:

- (Independence) Each party is elected independently (so multiple parties may be elected in the same round).
- (Fairness) The probability of electing each party is proportional to its relative computational power.
- (Pre-Commit Non-Equivocation Leadership announcement) In each round, each party commits to an action and the oracle probabilistically elects parties and announces them and their action.

We name an oracle with these properties a Fair, Pre-Commit, Non-Equivocation, Leader Elections (PCNELE) Oracle. The first PCNELE property, Independence, implies that in some rounds it may elect multiple leaders. This is typically tolerated in distributed protocols, as long as sufficiently often just one leader is elected.

The second one, Fairness, is important for many distributed protocols, e.g., for load balancing or contention resolution. It is crucial for Bitcoin's setting because winning an Oracle election has economical value.

The last, Pre-Commit Non-Equivocation, captures two unique functionalities. In a Byzantine fault model, just electing a leader is not enough because we want the leader to be able to add just a *single* block (and not be able to send different blocks to different parties). Moreover we would like the potential leader to *commit* to its single block content *before* the election

winner(s) are announced. In particular this means the leader cannot equivocate by sending two different messages to different parties.

Note that PCNELE is much stronger than the classic Ω Oracle that is sufficient for consensus in an asynchronous environment. This oracle is a key building block of the Nakamoto Consensus protocol. Getting the full protocol requires just one more idea: each new block will contain a reference to a previous block and the protocol advices to connect the new block so it forms the longest chain. Before we get there, we discuss a concrete PCNELE implementation via crypto-puzzles.

Implementing PCNELE using crypto-puzzles. In the CTA model, we can implement the FPCNELE oracle using cryptographic puzzles. Such a puzzle needs to have three properties:

- (Pre-commit) Solving the puzzle requires to commit to a specific *block*.
- (Public verifiability) The solver of the puzzle can generate a *solution certificate* of his correct solution and his committed block. This certificate can be efficiently publicly verified.
- (Fairness) the probability of solving the puzzle (generating a solution certificate) at any given round is proportional to the computational power of the party.

Note that this definition inherently assumes a round based model (or a partially synchronous model).

We now detail a concrete protocol based on a very simple cryptographic puzzle (a simplification of the Bitcoin cryptographic puzzle). We use two parameters: P which is the base of the puzzle and H which is the hardness parameter. For party with a public key i and a desired action o to solve the puzzle it must find a number *nonce* such that

$$SHA1(P||i||o||nonce) < H$$

Here we use SHA1 as an example of a secure hash function. Pre-commit is obtained because the action o is embedded into the puzzle and it is computationally hard to find a collision with another action o'. Public verifiability is obtained, because once the party discovers an adequate nonce, then it can publish it and anyone can verify solution. Fairness is obtained under the assumption that the cryptographic hash function is essentially a random function. So the only solution strategy is use brute force to find the nonce.

The parameter H needs to be tuned relative to the total amount of computational power. A large value of H will cause many parties to solve the puzzle in the same round, causing contention. A small value of H may cause the expected time it takes to elect a leader to be too long. Note that the election process essentially induces a Poisson process on elected parties whose parameter depends on H.

2.4 The Nakamoto Consensus protocol - Longest Fork Wins

At its core, Nakamoto Consensus (NC) is a protocol that implements a replication of a Blockchain by using two elegant and powerful ideas: PoW and a longest fork win (LFW) strategy.

The LFW strategy works as follows. When pre-committing a block content, a party must also pre-commit on a reference to a previous block. The LFW rule is to always choose to reference the longest chain.

We begin by providing an abstract solution that assumes access to an abstract FPCNELE Oracle of the previous section. We then define what problem this solution solves. Finally we detail a concrete example that follows the Bitcoin protocol.

An abstract NC protocol: Implementing Blockchain replication We now focus on implementing Blockchain replication assuming access to a FPCNELE Oracle.

The abstract NC protocol has just two principles:

- (New Block via FPCNELE Oracle): the Oracles allows each elected leader to announce a single pre-committed new block.
- (Longest Fork Wins): non-faulty leaders will connect their pre-committed block to the leaf block which forms the longest path from the genesis block.

Concretely, in each round, parties pre-commit to the FPCNELE Oracle which block they want to add and to which existing block they want to add it to. The Oracle then announces the set of parties that are elected (possibly none, possibly more than one) and the blocks of the elected parties are announced.

Now that we defined the abstract NC protocol, let's formally define what problem it solves.

Blockchain replication The core goal of state-machine replication (SMR) is to form a growing log of commands. A new command is appended to the tail of the log by a consensus decision, and does not modify anything before the tail. We will return to the classical SMR problem in the next section. Here, we view an even more abstract graph theoretic model.

We can view a log of commands as a dynamically growing *directed path*. A new command is added by adding a new node that points to the previous last node of the path. To define a *Blockchain replication protocol* we need to deal with potential forks: instances where there is (some) uncertainty about which command it decides on. For example, there may be several competing blocks that all point to the same parent block. To model this we need to extend the dynamically growing directed path abstraction to a more general *dynamic direct acyclic graph* model. We will call this graph G and call the root node g of G the "genesis" node. Given any existing leaf y we can extend y with a new node x by announcing the edge $y \leftarrow x$. We call this operation "announcing a new block".

We note that while G can be an arbitrary DAG, the goal of a Blockchain protocol is to force G to be as similar as possible to a directed path.

Blockchain Replication Properties. The goal of the Nakamoto Consensus protocol is to implement a *Blockchain replication protocol* in the computational threshold adversary model.

To this end we define four desired properties:

- (Uniqueness) there is a unique deterministic function $L(G)$ to extract a single path $P = g, a_1, a_2, a_3, \ldots, a_k$ from G. Typically $L(G)$ is the longest path in G from the genesis (with some tie-breaker).

- (Liveness) the path $L(G)$ is constantly growing (this often depends on the synchrony assumptions, for example: that a new block is added in expectation every 10 minutes).
- (Safety) If $P = g, a_1, a_2, a_3, \ldots, a_k = L(G)$ is a path of length k for G and G grows to become G' then for any $a_i \in P$ the probability that $a_i \notin L(G')$ is exponentially decreasing proportional to $k - i$. This is the famous "burying" property: the deeper a block is buried in the path P the harder it is to revoke it. In bitcoin, a block is typically considered "committed" if its buried behind a chain if 6 newer blocks (= one hour's worth of computational puzzle solving).
- (Fairness) the proportion of nodes in P that belong to non-faulty parties is at least proportional to their relative computational power.

We note that uniqueness is typically obtained by defining $L(G)$ as the longest path in G. If there are several paths of maximal length then we can deterministically choose one of them. Using a randomized tie breaker is also an option and has certain advantages (improved fairness and resilience to selfish mining) and disadvantages (potentially slower conversion).

The liveness and fairness properties are natural: we want the unique chain to grow and we want the proportion of blocks to represent the proportion of computational power of the parties.

The safety property is somewhat subtle. In order to make sure that a block is committed and will not be removed with high probability one needs to wait until the block is "buried" deep enough in the path P.

As we will later discuss, a solution for Blockchain replication can be trivially used to solve state-machine replication: simply define the state-machine as the operations in $P = L(G)$ after removing the k most recent nodes in P. Note that this implementation provides safety only with high probability.

Analyzing abstract NC If non-faulty parties follow the LFW principle, then it can be shown that indeed the properties of Blockchain replication are held.

While Liveness and Fairness seem to follow almost directly from the FPCNELE Oracle properties, again Safety is more subtle. Roughly speaking, the reason that a block a_i that is "buried" k blocks deep will not be replaced is that the probability that the adversary manages to fork the chain from a_{i-1} and create an alternate longer chain of length $k + 1$ is exponentially small as a function of k. The exponent of this property is a function of how far the adversary is from controlling half of the computational resources.

After defining an abstract NC protocol (that uses an Oracle) and defining what problem it solves we now describe a very simply concrete NC protocol.

2.5 Nakamoto Consensus - A concrete protocol

A concrete Nakamoto consensus protocol can be written in a single line:

- Given a party i that wants to commit a block with transactions o, let $P = a_1, \ldots, a_k$ be the longest path in G then party i attempts to solve the puzzle that will allow it to announce the $a_k \leftarrow b$ where b is a new block that contains i public key, and o.

  Concretely, find the nonce such that

$$SHA1(a_k||i||o||nonce) < H$$

The simple, single line protocol is a simplified version that captures the central idea behind the Bitcoin's Blockchain consensus protocol. At its core it uses Proof-of-Work to implement a Fair Leader Election Oracle that provides non-equivocation and pre-commitment. Non-faulty leaders are expected to always use the Longest Fork Wins rule and extend the longest path. This in turn implies that the non-faulty parties manage to implement a Blockchain replication protocol. Which in turn can be used to implement a state-machine replication protocol (where safety is guaranteed with high probability). A rigorous and detailed analysis of this approach appears in [20, 34].

3 Relating NC to BFT

When an NC leader announces a block $a_k \leftarrow b$, it implicitly announces not just the new block b, but also any block in $\{a_1, ..., a_k\}$ missing from G. By the Longest Fork Wins principle, this leader "proposal" wins if the path $a_1, ..., a_k, b$ is $L(G)$, i.e., the longest path in G. As mentioned above, this is the NC solution to the classical state-machine replication (SMR) problem of reaching agreement on a *growing log of commands*.

Relating this to the classical SMR literature, there is a common theme where proposals are prioritized by leader *ranks*: A proposal by a *highest ranking leader* is accepted and forces the leader's log-prefix. In the BFT literature, replication consistency is maintained by two principles:

- (Non-equivocation) leaders are prevented from *equivocating*, so that there is only one possible proposal per leader per rank
- (Proposal-safety) a (higher-ranking) proposal may extend, but not modify, any lower-ranking committed log prefix.

NC and BFT differ in how they accomplish these two key principles, and consequently, in BFT having instant finality and NC not having it.

We now put aside NC and discuss the foundations of replicated services. We will get back to relating the two approaches later.

SMR Problem Model. We start with a brief overview of state-machine replication (SMR) and related terminology. The State-Machine-Replication (SMR) approach [28, 36] is well known paradigm for building a fault-tolerant service, that linearizes all updates as if they occur one after another [23]. Driving an SMR service is a sequence of consensus decisions on a *growing log* of state-machine operations. The parties in the protocol are called *replicas*, and each state-machine *replica* executes the log of operations deterministically, arriving at a consistent replicated state. Here we adopt the traditional Threshold Adversary model (see §2). In this model, a threshold of the replica set may suffer certain failures; below, we discuss both benign failures and arbitrary corruptions (Byzantine failures). For simplicity, we neglect clients and the details by which requests arrive at replicas, and simply assume replicas have their own inputs.

Focusing on the consensus core, the key challenge is to repeatedly reach agreement decisions among the replicas on extending the log with a new command:

- (Agreement) There is agreement on a sequence of decisions among all correct replicas.
- (Validity) Committed decisions form a monotonically growing log of commands. Each command has been proposed by a replica. (In particular, in the Byzantine failure case, each command must carry some replica's signature.)
- (Liveness) If a correct replica proposes a command, then eventually some decision is committed.

3.1 Asynchronous benign framework with $N = 2F + 1$

We start with the benign-failure model, which allows us to focus on Safety; we get back to dealing with the threat of leader equivocation later, in the Byzantine-failure model.

We present a framework that borrows from the asynchronous consensus solution introduced by Dwork, Lynch and Stockmeyer in [16], and widely employed for SMR, e.g., in Viewstamped Replication [32], Paxos [29], Raft [33], and others. The DLS framework has the desirable property that Agreement is kept under all scenarios, including asynchrony; liveness depends on successfully electing a correct leader with timely communication channels to replicas.

A key concept of the framework is an explicit ranking among proposals. In DLS, ranks are termed *phases*, in Paxos, they are called *ballots*, and in VR, *views*. Here we use *views*. Replicas all start with an initial view, and progress from one view to the next. Commands are accepted in the highest view to have started. In some views, a decision will be reached to extend the log-prefix by one slot; other views will expire without a decision. Liveness relies on having a constant fraction of the views with a good and timely leader.

Decision values are monotonically increasing: Once a certain prefix becomes committed in a view, higher views can only extend it, but no slot in the committed log-prefix may ever be reverted. Therefore, once a certain prefix is committed, it becomes ready for execution by the replicas.

Algorithm 1 provides a breakdown of the framework into five abstract components: VIEW, a scheme for prioritizing proposals; WEDGE, a mechanism for starting new views; SAFE, a mechanism for collecting information about lower views and picking a safe value to propose based on responses; ACCEPT, a mechanism for making an accepted leader proposal durable; and DECIDE, a predicate for committing a decision.

Briefly, in each view there is a single designated *leader* (in fact, we typically identify the leader with the view-number). The other replicas are called *acceptors*, they accept at most one proposed value in the view. A replica moves to a higher view if a local timer expires.

A leader in a view ensures Validity by collecting information from a quorum that intersects every commit quorum in lower views in at least one correct acceptor. The leader then picks a safe log-prefix to propose, and extends it by one slot. Specifically, in a benign settings, the algorithm works for $N = 2F + 1$ replicas and quorums of size $N - F$. The leader collects STATUS messages from a quorum of $N - F$ acceptors, thus intersecting the quorums of all previous views. Among all the accepted prefixes reported to it, the leader picks the one accepted at the highest view, if any, or an empty prefix if none. It extends the log-prefix with its own input command, and proposes it to replicas.

Pipelining. Practical SMR solutions, such as VR/Paxos/Raft, let the leader of the current view drive repeated extensions to the growing log. This enables an optimized pipeline of proposals: A stable leader avoids waiting for the current view to expire for each command, and does not need to collect status messages. In this tutorial, for pedagogical reasons we will ignore this optimization (for a discussion of this optimization, see [11]).

Correctness In a nutshell, this protocol guarantees Agreement because a leader proposes only safe proposals: A value is safe to propose in a view if no lower view can ever decide a conflicting value.

As for liveness, the protocol makes a decision as soon as a good view is activated, meaning that the view-leader has timely communication with a majority quorum, none of which times out and moves to a higher view.

Algorithm 1 Benign SMR solution skeleton with $N = 2F + 1$.

VIEW Initially, each replica starts in view 0. A replica starts view $v + 1$ if view v expires with no progress, or if it receives any message from a view higher than its current view. The leader for view v is the replica whose ID modulo N equals the view number. We denote it by $L(v)$.[1]

WEDGE An acceptor Q, upon starting view v, sends $L(v)$ a message (STATUS, v, Q, H), where H is the proposal and view number in a maximal view for which Q ever received a proposal, or $\perp$ if none received. After moving to view v, an acceptor rejects messages from views lower than v.

SAFE $L(v)$ waits for $N - F$ status responses (STATUS, v, Q, H_Q). It picks a safe value S to propose. S is the status value whose view is highest among the STATUS responses collected, if any, or an empty log.

ACCEPT $L(v)$ extends S with its own input value, and sends the extended prefix $S*$ to all replicas a message (PROPOSE, v, $S*$).

An acceptor Q in view v, upon receiving a (PROPOSE, v, $S*$) message from $L(v)$, records v, $S*$ as the highest proposal it accepted, and sends (ACK, v, Q) to the leader.

DECIDE When $N-F$ acceptors of any view v accepted a proposal S it becomes the committed decision. The leader, or anyone learning this decision, broadcasts a (DECIDE, v, $S*$) notification.

3.2 Asynchronous Byzantine framework with $N = 5F + 1$

The above solution works in a model where the adversary can only cause crash failures. For the Byzantine model where the adversary can perform arbitrary actions on the faulty parties we need to make a few adjustments. First, all messages are signed, to prevent spoofing and to enable forwarding messages on behalf of others as proofs of safety and validity in various steps.

Second, the quorums of the SAFE mechanism, which are used for collecting information about lower views and picking a safe value to propose based on responses, need to be adjusted to guarantee intersection in sufficiently many non-faulty replicas [31]. Third, the ACCEPT component requires a mechanism that prevents a leader from equivocating and sending conflicting proposals to different parties.

The first solution we illustrate addresses these challenges simply by increasing quorum intersection to $3F + 1$. Consequently, this solution requires $N = 5F + 1$, and does not have optimal resilience. Nevertheless, it is a simple adaptation of Algorithm 1, and has the benefit of incurring linear communication complexity. Similar $5F + 1$ schemes appeared in [27, 1]. Algorithm 2 below highlights the modified parts.

Algorithm 2 Byzantine SMR solution skeleton with $N = 5F + 1$.

VIEW Initially, each replica starts in view 0. A replica starts view $v + 1$ if view v expires with no progress, or if it receives $F + 1$ messages (directly or indirectly) from a view higher than its current view. The leader for view v is the replica whose ID modulo N equals the view number. We denote it by $L(v)$.

WEDGE An acceptor Q, upon starting view v, sends $L(v)$ a message (STATUS, v, Q, H), where H is the proposal and view number in a maximal view for which Q ever received a proposal, or $\perp$ if none received. After moving to view v, an acceptor rejects messages from views lower than v.

SAFE $L(v)$ waits for $N - F$ status responses (STATUS, v, Q, H_Q). It picks a safe value S to propose. S is the proposal whose view is highest such that $2F + 1$ replicas accepted it in the view, if any, or $\perp$ if none received.

ACCEPT $L(v)$ extends S with its own input value, and sends the extended prefix $S*$ to all replicas a message (PROPOSE, v, $S*$). It attaches to its proposal a proof of safety, e.g., the collection of STATUS messages.

An acceptor Q in view v, upon receiving a (PROPOSE, v, $S*$) message from $L(v)$ for the first time, verifies the proposal validity and then records v, $S*$ as the highest proposal it accepted, and sends (ACK, v, Q) to the leader.

DECIDE When $N-F$ acceptors of any view v accepted a proposal S it becomes the committed decision. The leader, or anyone learning this decision, broadcasts a (DECIDE, v, S) notification.

A brief note about correctness of this Byzantine protocol. It guarantees agreement because each two quorums intersect in $3F + 1$ replicas, and $2F + 1$ of them are correct. Therefore, if a

decision is made in a view, a higher view intersects it in $2F + 1$ responses. Furthermore, in a system of $N = 5F + 1$, no other value can appear $3F + 1$ times. Similar to the benign case, the protocol is live if a correct leader has timely links with a quorum.

3.3 Synchronous Byzantine framework with $N = 3F + 1$

We can adapt Algorithm 2 to a synchronous settings, such that the resulting synchronous algorithm requires only $N = 3F + 1$ replicas, instead of $N = 5F + 1$. We do not repeat the algorithm, and instead we just highlight the adaptations.

In a synchronous setting, a party can collect messages from all non-faulty replicas by waiting the appropriate number of maximal transmission durations.

The leader sends a signed PROPOSE. A replica accepts the first PROPOSE message it receives and "echoes" it–adding its own signature–to all other replicas in a PROPOSE message.

After it accepts a proposal, a replica waits for echoes from all other replicas. A replica DECIDES if it hears $N - F$ PROPOSE messages with the same value and does not hear any PROPOSE message with a different value.

For a decision to be safe, the SAFE component requires a new leader to choose the value whose view is highest in which $F + 1$ replicas accepted an identical value, if any, or $\perp$ if none received.

Briefly, this algorithm maintains Agreement because a decision is reached in a round in which all correct replicas accept some proposal and no correct replica accepts a different proposal. Because correct replicas always respond in time, a leader of a higher round picking among STATUS messages the highest view in which $F + 1$ replicas accepted an identical value is guaranteed to choose this value as the only safe possibility.

3.4 Byzantine frameworks with optimal resilience

The two Byzantine protocols above do not have optimal resilience. To obtain a protocol for $N = 3F + 1$ for the asynchronous model, and likewise, $N = 2F + 1$ for the synchronous model, the trick is to replace the leader's broadcast with a protocol that provides two guarantees. One, the leader cannot equivocate. And two, every non-faulty replica that echoes a leader proposal can prove that it is unique. Obtaining these two properties typically requires replacing the trivial leader broadcast with a two-round protocol [14].

Asynchronous Byzantine framework with $N = 3F + 1$: In the asynchronous model, the leader protocol works in two-phases as follows. In the first phase, the leader sends a signed proposal tagged PRE-PROPOSE. Replicas echo–adding their own signature–the first leader's PRE-PROPOSE message they receive either directly from the leader, or in a PRE-PROPOSE message they receive from other replicas. In the second phase, upon receiving the same value in either $2F + 1$ PRE-PROPOSE messages, or $F + 1$ PROPOSE messages, replicas echo with a signed PROPOSE message and accept the proposal.

Upon accepting a PROPOSE value from the 2-phase broadcast protocol, replicas proceed to process the message as a leader proposal. Specifically, a decision is reached in a view (as usual) upon receiving $N - F$ PROPOSE messages for a value.

For a decision to be safe, the SAFE component requires a new leader to choose the value whose view is highest, such that either (i) $F + 1$ replicas sent a PROPOSE message with the

value, if any, or (ii) $2F + 1$ replicas sent a PRE-PROPOSE for it, if any, or (iii) $F + 1$ replicas sent a PRE-PROPOSE for it, if any. It chooses $\perp$ if non such value is received.

For further details of folding this leader broadcast protocol into the SMR full protocol, we refer to the PBFT protocol of Castro and Liskov [10]. Note that, the communication complexity here is quadratic, and matches the communication lower bound of Dolev, Reischuk and Strong [15].

Synchronous Byzantine framework with $N = 2F + 1$: In synchronous settings, we can implement the leader's broadcast protocol using two all-to-all exchanges as follows.

In the first phase, the leader broadcasts a PRE-PROPOSE value. Replicas echo to all other replicas–adding their own signature–the first leader PRE-PROPOSE message they receive.

In the second phase, a replica accepts a value if all PRE-PROPOSE echoes (of which there are at least $F + 1$) match. Note that, due to synchrony, if two correct replicas echo conflicting PRE-PROPOSE values, no correct replica will accept any value.

Upon accepting a value, a replica broadcasts to all other replicas a PROPOSE messages that carries the value it accepted, along with a proof of its validity, e.g., $F+1$ signed PRE-PROPOSE echoes.

It should be obvious from the above discussion that if any value is proposed by a correct replica, then it is unique. Furthermore, in that case, there may be no valid PROPOSE with a conflicting value. Therefore, a decision can be reached in a view upon receiving $F + 1$ PROPOSE messages carrying an identical value.

For this decision to be safe, upon starting a new view a replica broadcasts a PRE-STATUS message containing the last value it accepted, if any, along with a proof of its safety. It waits for all other PRE-STATUS messages and keeps among the valid ones the one whose view is highest. The SAFE component requires a new leader to choose the value whose view is highest which a *single* replica proposed, if any, or $\perp$ if either conflicting ones received or none received. The leader attaches to a PRE-PROPOSE messages a proof of safety, e.g., $F + 1$ STATUS messages.

Briefly, this algorithm maintains Agreement because if any correct replica sends PROPOSE, no conflicting valid PROPOSE exists. The leader cannot hide the proposed value because all correct replicas will receive a PRE-STATUS message about it.

This synchronous $N = 2F + 1$ paradigm appears in [2]. A slightly different leader-based solution paradigm called XFT, which guarantees safety against Byzantine failures in synchronous settings, is given by Liu et al. in [30]. A framework for several of these trade-offs is provided in [5].

3.5 Back to Nakamoto Consensus

We now get back to the world of *parties* that solve computation puzzles instead of replicas, and a Computational Threshold Adversary instead of a Threshold Adversary.

At first sight, Nakamoto Consensus (NC) seems very different than traditional SMR protocols. Instead of an explicit safe value selection scheme it just uses Longest Fork Wins to prioritize the currently longest "proposed" chain. Instead of using explicit quorum to guarantee that decision persists, NC uses synchrony assumptions and cryptographic puzzles to progressively make decisions harder to revoke.

Nevertheless in this section we attempt to view NC in the SMR framework we have just detailed. This exercise highlights many of the similarities. Algorithm 3 illustrates how NC (almost) implements the necessary ingredients we characterized above.

In NC, the notion of a view number (and implicit proposal rank) translates into chain-length; the longer the chain, the higher the rank (with ties broken by some deterministic rule). Like BFT, initially, each party starts in view 0 and moves from one view to the next. A leader of view v is a party solving a puzzle for a chain of length $v-1$, which it announces in a PROPOSE message sent to everyone. The Computational Threshold adversary captures the guarantee of having a constant fraction of the views with a good and timely leader. Indeed, in NC, if an adversary takes control of more than 50% of the computation power in the network, it can theoretically prevent progress: By forking blocks buried at arbitrary level, it can prevent any block from ever becoming finalized.

Similar to BFT solutions, the NC PROPOSE message is self-validating and causes other parties to move to view v when they receive this announcement, and subsequently reject messages from lower views. Different from BFT, there is no explicit collection of STATUS responses, nor explicit proof of safety of the proposal. Nevertheless, NC obtains Agreement (with high probability), and we now explain the relation to STATUS messages and quorums.

The key observation is that a party acceptance is implicit by the collaborative work represented in a puzzle solution. More specifically, recall that we model a Fair PCNELE Oracle model in §2 as choosing only one (or a few) blocks to extend G out of all the parties attempting to. Since all parties wish their proposal to win longest path of G, a block announcement $a_{v-1} \leftarrow b$ that extends the current longest path $a_1, ..., a_{v-1}$ in G to length v represents the collective "acceptance" of this path by many parties presenting proposed blocks to the Oracle. Indeed, if a party tried to introduce a path that replaces the last ϵ blocks of $L(G)$, it would need to win the oracle's choice ϵ times. This occurs with probability that decays exponentially with ϵ. Thus, the PCNELE selection policy incentivizes parties to accept G and extend $L(G)$, rather than replacing any part of it.

In summary, a PoW $a_{v-1} \leftarrow b$, although not a strict proof of acceptance by a quorum of the parties, is a statistical tallying of their acceptances of $a_1, ..., a_{v-1}$. Hence, while Validity is not strictly guaranteed, the probability of replacing a block buried ϵ levels deep is exponentially diminishing with ϵ. As a corollary, Agreement on blocks buried k level deep in $L(G)$ holds with high probability.

Algorithm 3 NC in the view of the Threshold Adversary framework.

VIEW Initially, each party starts in view 0. By the "longest-chain-wins" principle, upon receiving an announcement that extends $L(G)$ to length v, a party moves to view v if it is currently in a lower view.

A party becomes a leader $L(v)$ for view v when solves the puzzle that allows it to announce into G a block $a_{v-1} \leftarrow b$, such that b forms a chain $a_1 \leftarrow ... \leftarrow a_{v-1} \leftarrow b$ of length v. Note that this leader is not necessarily unique, so there may be multiple leaders for view v, and multiple proposals with the same view.

WEDGE Upon starting a new view, a party chooses to work on the puzzle at the end of $L(G)$. By the "longest-chain-wins" principle, after moving to view v, a party rejects messages from views lower than v.

SAFE Successfully solving the puzzle at the end of $L(G)$ implicitly and statistically represents acceptance of $L(G)$ by a large fraction of parties.

ACCEPT A leader $L(v)$ announces to all parties a block $a_{v-1} \leftarrow b$ that extends $L(G)$ to length v. A party chooses to work on the puzzle at the end of $L(G)$, implicitly conveying acceptance.

DECIDE A party in view v may decide on the block at position $v - \epsilon$ in $L(G)$, the longest chain in G. With probability proportional to ϵ, this block will not be overturned by any future insertions to G.

4 Combining NC with BFT

In this section we discuss emerging approaches to combine NC with BFT. Such hybrid solutions address the deficiencies in either arena, and bring the combined benefits of both worlds.

We begin by underscoring challenges of either paradigm.

4.1 Blockchain replication challenges

While Blockchain replication can be used to implement state-machine replication, this implementation has several limitations.

Lack of finality Blockchain replication suffers from a long-term deficiency where there is always the risk that a block will be un-committed. In fact the formal safety seems to require infinite executions and to assume that the relative minority computational bound of the adversary will remain true till infinity. In practice this raises many concerns: what if after some safe period an adversary acquires more than half of the relative computational power and reverts all the transactions?

Commit Latency Blockchain replication suffers from a short-term deficiency. Even if we assume infinite execution and are willing to assume that a block that is buried by an hour's worth of cryptographic puzzles is completely safe, this still means we need to wait an hour! In fact one of the biggest advantages of BFT protocols is that they provide "instant finality". This is a property that the NC does not achieve.

Selfish Mining This is an attack on the fairness property first suggested by Eyal and Sirer [19]. This attack is related to the incentives behind publishing PoW puzzles immediately when they are solved.

4.2 BFT State-machine replication challenges

State-machine replication with Byzantine fault tolerance brings many advantages. Unlike NC, it does not suffer from lack of finality. On the contrary, once a block is committed it can never be revoked. Moreover, modern BFT SMR systems have very low latency and high throughput. In particular these systems are often tuned to provide high performance in the common, failure-free case. Even in a WAN deployment with tens of servers these systems typically manage to commit hundreds (if not thousands) of operations per second in good network conditions and failure free executions.

The permissionless model of NC allows for a much larger set of replicas. It is commonly believed that there are at least several thousands of miners implementing the BitCoin NC replication protocol.

While BFT SMR protocols seem to work well for few tens of servers, scaling these solutions to hundreds or even thousands of replicas raises new challenges.

4.3 Bridging the two paradigms

We proceed to mention a few approaches to combine both the NC and BFT techniques to overcome the above challenges.

The Ethereum Casper [9] approach harnesses BFT to deal with the lack of finality in NC. Casper sets up a trusted committee of validators, who may be selected by placing a security deposit. The committee post-validates NC blocks (buried to a certain depth) via a BFT protocol. Only after a path is validated by the committee it is considered committed. In this approach, the NC chain can be viewed as a client sending a stream of requests to a permissioned BFT engine. Note that due to the slow cadence of requests (e.g., an Ethereum average block time is 14 seconds), the BFT protocol can work in essentially synchronous mode.

Another approach uses NC to elect a rotating committee, and then the committee is responsible for deciding on blocks to add to the chain. The first step in this approach was BitCoin-NG [18], they suggest that after a leader is elected it continues to perform micro-transactions. ByzCoin [26] took this idea a step further, and defined a committee of size C simply as the last C elements in the NC chain. ByzCoin suggests that this committee then proceeds to use PBFT to drive micro-transactions. One challenge is that committee members themselves may not be finalized. To remedy this, a similar approach is taken by Hybrid Consensus [35], where the committee C is chosen as the members that are buried $2C$ to $C + 1$ in the NC chain. In addition to finality of decisions, the benefit of this approach is that the BFT engine provides faster turnaround of (finalized) decisions.

A third approach introduced in Solidus [6] builds the Blockchain itself by consensus and does not use LFW at all. It uses PoW as a way to both mint new currency and rotate a committee. A newly minted block does **not** win by LFW, instead it needs to go through consensus approval by the existing committee in order to be added to the Blockchain. Once a block is added to the chain, it also rotates the committee (similar to Byzcoin and HC), i.e., the new element gets admitted to the committee, and the oldest element retired. Like the above approaches, Solidus provides finality and low latency. In addition, in Solidus the details of *reconfiguring* the committee members are weaved into the Blockchain protocol and are based on traditional SMR-BFT reconfiguration approaches.

Scaling Byzantine agreement through a randomized sample of a sub-committee is, in itself, a known idea (see, e.g., [25]). A vulnerability of all committee-based approaches is that an adversary can mount a targeted attack on the (smaller) set of committee members. Cryptographic techniques were introduced in [24] to hide the committee selection against such a *rushing adversary*. More recently, Algorand [21] uses Proof-of-Stake coupled with verifiable random functions (VRF) to secure a randomized selection of committee members in permissionless setting. Algorand further mitigates vulnerability by having each committee determine only one block, and converging in one communication step by each member.

5 Concluding Remarks

In this tutorial, we discussed the foundational aspect of Blockchain as a Byzantine fault-tolerant state-machine-replication engine.

We first provided a foundational breakdown of NC as a randomized protocol in the synchronous settings. We introduced the abstractions of a randomized oracle, that selects a small number of requests to extend a directed graph. We then expressed NC as a simple combination

of two principles, request to extend the longest chain in the graph, and accept the longest chain as winning.

We then introduced a general framework for classical BFT solutions that preserves two key safety properties: Validity and Agreement. It builds around a view by view solution that explicitly collect information from a quorum of parties, and determines a proper way to the log from one view to the next. We used the framework to relate NC to classical BFT solutions. The NC oracle, by admitting only a small selection of requested block additions, implicitly performs approximate quorum collection.

An exciting emerging direction is to bring the two worlds together. We provided a glimpse into several ideas for "hybrid" solutions.

As discussed in the introduction, the SMR engine of the Blockchain provides only the base layer for the Blockchain stack. Future tutorials may cover the smart contract layer, where there are fascinating connections between this service logic layer and parallel advances in the area of removing trust from centralized implementations.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, October 2005.

[2] Ittai Abraham, Srini Devadas, Kartik Nayak, and Ling Ren. Brief announcement: Practical synchronous byzantine consensus. In *Proceedings of the Int'l Symposium on Distributed Computing*, October 2017.

[3] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: Robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 53–62, New York, NY, USA, 2006. ACM.

[4] Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Lower bounds on implementing robust and resilient mediators. In Ran Canetti, editor, *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008.*, volume 4948 of *Lecture Notes in Computer Science*, pages 302–319. Springer, 2008.

[5] Ittai Abraham and Dahlia Malkhi. BVP: Byzantine vertical paxos. `https://www.zurich.ibm.com/dccl/papers/abraham_dccl.pdf`, May 2016. Accessed: 2016-11-08.

[6] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR*, abs/1612.02916, 2016.

[7] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged Byzantine impostors. Technical Report YALEU/DCS/TR-1332, Yale University Department of Computer Science, July 2005.

[8] Adam Back. Hashcash - a denial of service counter-measure. Technical report, 2002.

[9] Vitalik Buterin. Casper version 1 implementation guide. Technical report, 2017.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[11] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[12] Wei Dai. B-money. Technical report, 1998.

[13] Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. In *13th IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[14] Danny Dolev. The byzantine generals strike again. Technical report, Stanford, CA, USA, 1981.

[15] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[17] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139–147, London, UK, UK, 1993. Springer-Verlag.

[18] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 45–59, Berkeley, CA, USA, 2016. USENIX Association.

[19] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454. Springer, 2014.

[20] Juan Garay and Leonardos Kiayias. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *In Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[22] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '90, pages 437–455, London, UK, UK, 1991. Springer-Verlag.

[23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[24] Valerie King and Jared Saia. Breaking the $o(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary. *J. ACM*, 58(4):18:1–18:24, July 2011.

[25] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 990–999, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics.

[26] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium*, pages 279–296. USENIX Association, 2016.

[27] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

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

[29] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[30] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[31] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, October 1998.

[32] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[33] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[34] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *EUROCRYPT (2)*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, 2017.

[35] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. *IACR Cryptology ePrint Archive*, 2016:917, 2016.

[36] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[37] Nick Szabo. Smart contracts: Building blocks for digital market. Technical report, 1996.

[38] Nick Szabo. Bit gold. Technical report, 2005.

Composition in State-based Replicated Data Types

Carlos Baquero[1], Paulo Sérgio Almeida[1],
Alcino Cunha[1], and Carla Ferreira[2]

[1]HASLab, INESC-TEC & Minho University, Portugal,
{cbm,psa,alcino}@di.uminho.pt
[2]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal,
carla.ferreira@fct.unl.pt

Abstract

Keeping replicated data strongly consistent is convenient when communication is fast and available. In internet-scale distributed systems the reality of high communication latencies and likelihood of partitions, leads developers to adopt more relaxed consistency models, such as eventual consistency. Conflict-free Replicated Data Types, bring structure to the design of eventually consistent data management solutions, by precisely describing the behaviour under concurrent updates and guarantying a path to reconciliation. This paper offers a survey of the mathematical structures that support state based multi-master replication with reconciliation, and shows how state structures and state transformations can be composed to provide data types that are now used in practice in many geo-replicated systems.

1 Introduction

Eventual consistency [25] is a relaxed, and highly available, data consistency model that is often the option of choice in internet-scale distributed systems. The common reasoning is that availability must be maintained, despite outages and partitioning, whereas delayed consistency is acceptable. Replicated data can be independently updated by multiple masters, allowing replicas to temporarily diverge [15], provided that they can eventually be reconciled into a common state. The notion of attaining eventual consistency when updates stop can be traced to R. H. Thomas in [23] *"By mutual consistency we mean that all copies converge to the same state and would be identical should update activity cease"*.

Reconciliation of divergent data has been studied for many years, with strong roots in databases [22, 23] and distributed file-systems [17, 18], often motivated by support of disconnected operation. However, reconciliation algorithms used to be ad-hoc and had to be devised by application layer programmers, typically lacking a sound basis that ensured their correctness and convergence properties. Alternatively, reconciliation can be left to the user, as in version control systems, thus any outcome is possible and reconciliation becomes non deterministic.

Conflict-free Replicated Data Types (CRDTs) [21] are motivated by modern demands from internet-scale systems [3, 14, 16, 24]. Within those systems, they currently serve millions of users world-wide, and bring a more grounded approach to the design of efficient and deterministic reconciliation solutions. They preserve the sequential semantics of the modeled data types, and present a choice among deterministic options when addressing concurrent changes. For instance, when facing concurrent insertion and removal of the same element in a set, different set concurrency semantics can lead to Add-wins or Remove-wins CRDT Sets. While the choice of best CRDT data type implementation is still left to the application designer, each data type is still assured to be correct with respect to its sequential and specific concurrent semantics.

CRDTs support two complementary designs: operation-based and state-based. Operation-based CRDTs require a middleware that provides reliable causal delivery to a known group of replicas, while state-based CRDTs usually only require access to globally unique identifiers and eschew membership information. Due to their additional flexibility, state-based CRDTs have a larger ratio of adoption in industry, and will be the focus of this study.

State-based CRDTs are rooted in the mathematical structure known as join-semilattices (which in this document we will abbreviate to simply lattices). These order structures ensure that: 1) the replicated states of the data types evolve and increase in a partial order in a precise way, as operations are applied, so that the new version subsumes the previous one; 2) all concurrent evolutions can be merged deterministically by the lattice join. In order to understand the building principles of state-based CRDTs it is necessary to understand the basic building blocks of lattices and how lattices can be composed.

In the following sections we will make a bridge linking classic results from order and lattice theory into state-based CRDT construction techniques. We will show how state evolves within a lattice; present several examples of concrete CRDTs; and when possible link them to concrete use cases. We envision two main readership goals: to provide a compact reference of constructions for the benefit of data type developers, and to possibly entice theoreticians to consider a new subject area for practical application of lattice theory.

2 From Sets to Lattices

In this context the most basic structure to define is a set of distinct values. An example is the set of vowels that can defined by extension as $\mathsf{vowels} \doteq \{\mathtt{a}, \mathtt{e}, \mathtt{i}, \mathtt{o}, \mathtt{u}\}$. Elements in a set have no specific order and they only need to be distinguishable.

A partially ordered set, usually known as poset, is a set equipped with a binary relation $\sqsubseteq$ which is reflexive, transitive and anti-symmetric. Given any elements o, p, q in a poset we have:

- (reflexive) $p \sqsubseteq p$
- (transitive) $o \sqsubseteq p \land p \sqsubseteq q \Rightarrow o \sqsubseteq q$
- (anti-symmetric) $p \sqsubseteq q \land q \sqsubseteq p \Rightarrow p = q$

As an example, we can build a poset over the set of vowels by ordering just two elements $\mathtt{a} \sqsubseteq \mathtt{u}$, while the remaining elements are left unordered. These unordered elements are called concurrent.

- (concurrent) $p \parallel q \iff \neg(p \sqsubseteq q \lor q \sqsubseteq p)$

In one extreme, we can build a poset with a total order on the set of vowels with $\mathtt{a} \sqsubseteq \mathtt{e} \sqsubseteq \mathtt{i} \sqsubseteq \mathtt{o} \sqsubseteq \mathtt{u}$. In this example we ordered all elements and thus created a chain, i.e. a set where for any two elements p, q we have either $p \sqsubseteq q$ or $q \sqsubseteq p$.

In the other extreme, we can leave all elements unordered and define a poset that is an antichain, where any two elements are always concurrent. E.g., for the vowels, defining $\sqsubseteq \doteq \{(\mathtt{a}, \mathtt{a}), (\mathtt{e}, \mathtt{e}), (\mathtt{i}, \mathtt{i}), (\mathtt{o}, \mathtt{o}), (\mathtt{u}, \mathtt{u})\}$.

Throughout the paper we will use simple typing rules to clarify how some structures can be obtained from others (by composition or simply by shedding some properties). For example, every poset is obviously also a set:

$$\frac{A : \mathsf{poset}}{A : \mathsf{set}}$$

Given a poset A and a subset S of A, an upper bound of S is an element of A that is greater than or equal to all elements of S. The *least upper bound*, if it exists, is an upper bound that is less than any other upper bound, and therefore, unique. Going back to the chain defined over the set of vowels ($\mathtt{a} \sqsubseteq \mathtt{e} \sqsubseteq \mathtt{i} \sqsubseteq \mathtt{o} \sqsubseteq \mathtt{u}$), considering the subset $\{\mathtt{a}, \mathtt{i}\}$, elements $\mathtt{i}$, $\mathtt{o}$, $\mathtt{u}$ are all upper bounds of the subset, while $\mathtt{i}$ is the least upper bound.

A given poset A is a lattice if there exists a least upper bound for any pair of elements p and q in A, written $p \sqcup q$, being $\sqcup$ called the *join* operator. By definition, this binary join satisfies the following properties:

- (idempotent) $p \sqcup p = p$
- (commutative) $p \sqcup q = q \sqcup p$
- (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$

We can generalise it to express the least upper bound of any non-empty finite set S in a lattice A as $\bigsqcup S$. Some properties of least upper bounds are:

- (upper bound) $o \sqsubseteq o \sqcup p$
- (least upper bound) $o \sqsubseteq p \land o \sqsubseteq q \Rightarrow o \sqsubseteq p \sqcup q$

There are posets where the join does not exist for all pairs of different elements; these are not lattices. For instance, an antichain is not a lattice, as the join of any pair of different elements does not exist. Another example is bit strings under prefix ordering (e.g., $01 \sqsubseteq 010$) where concurrent elements, e.g., $010 \parallel 100$, are not joinable.

However, having a set and any idempotent, commutative and associative binary operation, which can be called a join, we have a lattice, with the order induced by the join as $p \sqsubseteq q \iff p \sqcup q = q$.

$$\frac{A : \mathsf{lattice}}{A : \mathsf{poset}}$$

When implementing CRDTs, where all possible states must have a join, this can allow skipping the direct implementation of $\sqsubseteq$ and deriving it from $\sqcup$. However, for performance reasons, it might still be advisable to directly implement $\sqsubseteq$ when appropriate.

As will be presented in Section 4.5, a lattice can be obtained from any set A, by using the powerset $\mathcal{P}(\mathsf{A})$ as the supporting set and choosing the order to be set inclusion, which results in the join being set union. In our running example this would be the lattice defined by $\langle \mathcal{P}(\mathsf{vowels}), \subseteq, \cup \rangle$.

As another general rule, any totally ordered set, i.e., any chain, is a lattice, with the join being the maximum.

$$\frac{A : \mathsf{chain}}{A : \mathsf{lattice}}$$

For natural numbers we have the lattice $\langle \mathbb{N}, \leq_{\mathbb{N}}, \mathsf{max} \rangle$. These simple lattices, and others, can be found as building blocks for the Bloom^L system [7], a language supporting eventual consistency without coordination.

Some lattices have a least element, called the bottom element $\bot$. In these cases the least upper bound for the empty set $\bigsqcup \emptyset$ exists, and it is precisely $\bot$. Some properties are:

- (bottom) $\bot \sqsubseteq o$
- (identity) $\bot \sqcup o = o$

Some examples: the lattice formed by the powerset of a given set A has the empty set as bottom, $\langle \mathcal{P}(A), \subseteq, \cup, \emptyset \rangle$, and natural numbers have 0 as bottom. For those lattices that do not have a bottom, it is always possible to add an extra element as bottom, ordered before all others, obtaining a lattice with bottom. We will address this construction when talking about lattice composition by linear sums in Section 4.3. Trivially, lattices with bottom are lattices.

$$\frac{A : \mathsf{lattice}_\bot}{A : \mathsf{lattice}} \qquad \frac{A : \mathsf{chain}_\bot}{A : \mathsf{chain}}$$

2.1 Primitive Lattices

We now introduce a small set of lattices, that will be later useful to construct more complex structures by composition.

Singleton A single element, $\bot$.

$$\frac{}{\mathbb{1} : \mathsf{chain}_\bot}$$

$$\bot \sqsubseteq \bot \qquad \bot \sqcup \bot = \bot$$

Boolean Two elements $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$ in a chain, where join is logical $\vee$.

$$\frac{}{\mathbb{B} : \mathsf{chain}_\bot}$$

$$\mathsf{False} \sqsubseteq \mathsf{True} \qquad x \sqcup y = x \vee y \qquad \bot = \mathsf{False}$$

Naturals Natural numbers with maximum as join. We include the 0, thus $\mathbb{N} = \{0, 1, \ldots\}$.

$$\frac{}{\mathbb{N} : \mathsf{chain}_\bot}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \mathsf{max}(n, m) \qquad \bot = 0$$

Integers Integers with maximum as join.

$$\frac{}{\mathbb{Z} : \mathsf{chain}}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \mathsf{max}(n, m)$$

3 Inflations make CRDTs

State-based CRDTs can be specified by selecting a given lattice to model the state, and choosing the initial state usually as the lattice $\bot$, if there is one. Query operations evaluate an arbitrary function on the state and return a value. Mutation operations do not return values and can only change the state by *inflations*. An inflation f is an endofunction over A that for any value x in A returns an element greater than or equal to x:

- (inflation) $x \sqsubseteq f(x)$

It should be noticed that an inflation is not the same as a monotonic function, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. (We have noticed this confusion sometimes.) As an example, the function $f(x) = \frac{x}{2}$ on positive reals is monotonic but not an inflation. Inflations can be further classified as non-strict and strict inflations, where a strict inflation is one such that:

- (strict inflation) $x \sqsubset f(x)$

The rules concerning inflations are thus:

$$\frac{\forall x \in a \cdot x \sqsubseteq f(x)}{f : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{\forall x \in a \cdot x \sqsubset f(x)}{f : A \xrightarrow{\sqsubset} A}$$

$$\frac{f : A \xrightarrow{\sqsubset} A}{f : A \xrightarrow{\sqsubseteq} A}$$

A state that is only updated as a result of inflations over its current value will not be modified if joined with some past state: the new state always subsumes the older one. This has important practical implications: state can be transmitted

at-least-once across replicas, since duplicates have no impact. If an old duplicate arrives at a replica, even out of order with more recent states, joining it with the local state will be harmless (a no-op), as its effect will have already been incorporated, and there is no danger of 'going backwards' and losing more recent information.

3.1 Primitive Inflations

Similarly to the primitive lattices introduced above we can define some primitive inflations.

$$\mathsf{id}(x) = x \qquad \frac{}{\mathsf{id} : A \xrightarrow{\sqsubseteq} A}$$

$$\underline{\mathsf{True}}(x) = \mathsf{True} \qquad \frac{}{\underline{\mathsf{True}} : \mathbb{B} \xrightarrow{\sqsubseteq} \mathbb{B}}$$

$$\mathsf{succ}(x) = x + 1 \qquad \frac{}{\mathsf{succ} : \mathbb{N} \xrightarrow{\sqsubset} \mathbb{N}}$$

3.2 Sequential Composition

Inflations can be composed sequentially. As long as there is at least one strict inflation in the composition, we obtain a strict inflation.

$$(f \bullet g)(x) = f(g(x))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubset} A}{f \bullet g : A \xrightarrow{\sqsubset} A} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubset} A}$$

4 Lattice Compositions

Since we are interested in creating lattices we consider a few composition techniques that are known to derive lattices. While in some cases they build from other lattices, in others they can derive lattices from simpler structures.

4.1 Product

The product $\times$, or pair construction, derives a lattice formed by pairs of other lattices. It can be applied recursively and derive a composition from a sequence of lattices, where operations are applied in point-wise order.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \times B : \mathsf{lattice}}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

$$(x_1, y_1) \sqcup (x_2, y_2) = (x_1 \sqcup x_2, y_1 \sqcup y_2)$$

The construction also extends to lattices with bottom.

$$\frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}_\bot}{A \times B : \mathsf{lattice}_\bot}$$

$$\bot = (\bot, \bot)$$

As an example, the underlying lattice structure of a version vector [15] among three replica nodes is composable by $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ with $\bot = (0, 0, 0)$.

Bellow are the properties of inflations over products. A strict inflation on one of the components leads to an overall strict inflation.

$$(f \times g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubseteq} A \times B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B}$$

Classic causality based event ordering mechanisms can be described by lattices that evolve by strict inflations. Lamport scalar logical clocks [13] are described by the $\mathbb{N}$ lattice where each event number is generated locally by strict inflation, and received remote clocks are merged-in by join. Similarly, vector clocks share the same structure as version vectors, a product composition of $\mathbb{N}$ lattices, with local events generated by strict inflation of the local entry.

4.2 Lexicographic Product

The ⊠ construct builds a lexicographic order from its source lattices. Components to the left are more significant and, unless they are equal, they filter out further comparisons towards the right side.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}_\bot}{A \boxtimes B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}_\bot}{A \boxtimes B : \mathsf{lattice}_\bot}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \vee (x_1 = x_2 \wedge y_1 \sqsubseteq y_2)$$

$$(x_1, y_1) \sqcup (x_2, y_2) = \begin{cases} (x_1, y_1) & \textbf{if } x_2 \sqsubset x_1 \\ (x_2, y_2) & \textbf{if } x_1 \sqsubset x_2 \\ (x_1, y_1 \sqcup y_2) & \textbf{if } x_1 = x_2 \\ (x_1 \sqcup x_2, \bot) & \textbf{if } x_1 \parallel x_2 \end{cases}$$

$$\bot = (\bot, \bot)$$

If the left component is a chain, often the case in practical uses based on timestamps or scalar logical clocks, then the right one can be any lattice (without requiring ⊥) as the fourth clause of the join definition never applies.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{lattice}}{A \boxtimes B : \mathsf{lattice}}$$

And, if the right component is also a chain the composition is a chain.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{chain}}{A \boxtimes B : \mathsf{chain}}$$

Some properties of inflations on lexicographic products are:

$$(f \boxtimes g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubseteq} A \boxtimes B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \longrightarrow B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B}$$

Notice that if we apply a strict inflation to the left component, then the right component can be transformed by any function even if non inflationary. In practice this allows resetting the right component after strictly inflating the left; we will see this in Section 5.2 when building lexicographic counters.

The abstraction provided by the lexicographic product is at the core of many practical systems that use *last-writer-wins* approaches to manage concurrent data updates [11]. By using fine grained timestamps in the left side and keeping node clocks as closely synchronized as possible across system nodes, one can expect strict inflations on the left as timestamps increase with time. When merging, higher timestamp values will determine the outcome of the join.

4.3 Linear Sum

The next composition, linear sum $\oplus$, picks two lattices, left and right, and creates a new lattice where any element from the left lattice is always ordered as less than any element in the right lattice. In the resulting set the elements are tagged with a label that identifies from which source lattice they come from. i.e., $\mathsf{Left}\ a$ means that element a comes from the left lattice and is now named $\mathsf{Left}\ a$. Tagging also ensures that the sets supporting each lattice can have elements in common.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\bot \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}_\bot}$$

$$\begin{array}{ll} \mathsf{Left}\ x \sqsubseteq \mathsf{Left}\ y = x \sqsubseteq y & \mathsf{Left}\ x \sqcup \mathsf{Left}\ y = \mathsf{Left}\ (x \sqcup y) \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Right}\ y = x \sqsubseteq y & \mathsf{Right}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ (x \sqcup y) \\ \mathsf{Left}\ x \sqsubseteq \mathsf{Right}\ y = \mathsf{True} & \mathsf{Left}\ x \sqcup \mathsf{Right}\ y = \mathsf{Right}\ y \\ \mathsf{Right}\ x \sqsubseteq \mathsf{Left}\ y = \mathsf{False} & \mathsf{Right}\ x \sqcup \mathsf{Left}\ y = \mathsf{Right}\ x \end{array}$$

$$\bot = \mathsf{Left}\ \bot$$

A possible use of this construction is to add a $\bot$ to a lattice that did not had one. For instance $\mathbb{1} \oplus \mathbb{R}$ can add a special element, e.g. nil, that is ordered as less than any real number. The same construction can also be used to add a top element $\top$ to a lattice, that can act as a tombstone that stops lattice evolution.

Some properties of inflations on sums are:

$$\begin{array}{l} (f \oplus g)(\mathsf{Left}\ x) = \mathsf{Left}\ f(x) \\ (f \oplus g)(\mathsf{Right}\ x) = \mathsf{Right}\ g(x) \end{array}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubseteq} A \oplus B}$$

$$\frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubset} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubset} A \oplus B}$$

4.4 Functions and Maps

The function space $A \to B$ is a lattice, obtained by combining a set A with a lattice B, and using pointwise comparison and join.

$$\frac{A : \mathsf{set} \qquad B : \mathsf{lattice}}{A \to B : \mathsf{lattice}} \qquad \frac{A : \mathsf{set} \qquad B : \mathsf{lattice}_\bot}{A \to B : \mathsf{lattice}_\bot}$$

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x) \qquad (f \sqcup g)(x) = f(x) \sqcup g(x)$$

$$\bot(x) = \bot$$

Many CRDTs are based on partial functions, i.e., maps $K \hookrightarrow V$, where K is any set of keys, and V is any lattice with bottom. Such maps are sometimes used to (efficiently) represent total functions, assuming that keys which are not present in the map implicitly yield bottom.

$$m(k) = \begin{cases} v & \textbf{if } (k, v) \in m \\ \bot & \textbf{otherwise} \end{cases}$$

This view of maps as functions also allows us to reuse the respective definition for join.

$$\frac{K : \mathsf{set} \qquad V : \mathsf{lattice}_\bot}{K \hookrightarrow V : \mathsf{lattice}_\bot}$$

An example of a map from vowels to integers $\mathsf{vowels} \hookrightarrow \mathbb{N}$ is $m = \{\mathtt{a} \mapsto 3, \mathtt{i} \mapsto 5\}$. Viewing this map as function we could query for $m(\mathtt{u})$ which yields 0.

We now define some inflations over maps. The first applies an inflation to all values in the co-domain and thus inflates the whole map.

$$\mathsf{map}(f)(m) = \{(k, f(v)) \mid (k, v) \in m\}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{map}(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

The second applies an inflation to the value for a given key.

$$\mathsf{apply}_k(f)(m) = m\{k \mapsto f(m(k))\}$$

Note that if the key is missing the function is applied to $\bot$.

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

Having maps we can refine the modeling of fixed size vector clocks that was based on products of $\mathbb{N}$. A dynamic vector clock can be obtained by mapping node identifiers to $\mathbb{N}$ lattices, as described in [7].

4.5 Sets and Multisets

As we have seen in Section 2, given any $\mathsf{set}\ A$ it is possible to derive a lattice with bottom by using the set of all possible subsets, the powerset $\mathcal{P}(A)$.

The powerset can also be defined by a function that maps each set element to a boolean that states its presence in the subset. This composition is very general since it can produce a lattice with bottom from any set.

$$\mathcal{P}(A) \cong A \to \mathbb{B}$$

Given that $\mathbb{B}$ is a lattice with bottom, from the previous section we know that function $A \to \mathbb{B}$ is also a lattice with bottom. Moreover, the respective order relation and join operator are obtained by construction and are equivalent to the expected.

$$\frac{A : \mathsf{set}}{\mathcal{P}(A) : \mathsf{lattice}_\perp}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \perp = \{\}$$

A natural extension is to represent *multisets* by mapping the domain set to naturals, instead of booleans.

$$\frac{A : \mathsf{set}}{\mathcal{P}_m(A) : \mathsf{lattice}_\perp}$$

$$\mathcal{P}_m(A) \cong A \to \mathbb{N}$$

Once again, by the properties of lattice composition, we get that function space $A \to \mathbb{N}$ is a lattice with bottom and both the order relation and join operator are provided by construction.

Given that both $\mathbb{B}$ and $\mathbb{N}$ are lattices with bottom, actual CRDTs for sets and multisets use maps to represent functions, as discussed in the previous section. The generic inflations defined for maps can be used here to define an inflation that adds an element e to a given set s.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\underline{\mathsf{True}})(s)$$

Likewise, to add an element to a multiset one increments the element count, having a strict inflation.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\mathsf{succ})(s)$$

4.6 Maximal Elements

A *down-set* (or order ideal) D of a poset P, is downward closed set, according to $\sqsubseteq$, of elements in P; i.e., if $x \in D$ and $y \sqsubseteq x$, then $y \in D$. Down-sets are useful in many situations, e.g., to represent *causal histories* [19] of all events in the past, up to a given point. Down-sets are also closed under set union, which means that the set of down-sets $\mathcal{D}(A)$ of a poset A is a lattice with bottom (similarly to the powerset for sets), with the usual set inclusion for order and union for join.

$$\frac{A : \mathsf{poset}}{\mathcal{D}(A) : \mathsf{lattice}_\perp}$$

But as they tend to get very large, they are used more as a modelling device, than as an actual construct in implementations. However, a down-set can be more compactly represented by the set of its maximal elements, which is an antichain.

$$\mathsf{maximal}(S) = \{x \in S \mid \nexists y \in S \cdot x \sqsubset y\}$$

This means that, starting from a poset A, we can obtain a lattice with bottom, isomorphic to $\mathcal{D}(A)$, which we call $\mathcal{M}(A)$: the lattice of maximal elements.

$$\mathcal{M}(A) = \{\mathsf{maximal}(S) \mid S \in \mathcal{P}(A)\}$$

$$\frac{A : \mathsf{poset}}{\mathcal{M}(A) : \mathsf{lattice}_\perp}$$

$$\mathcal{M}(A) \cong \mathcal{D}(A)$$

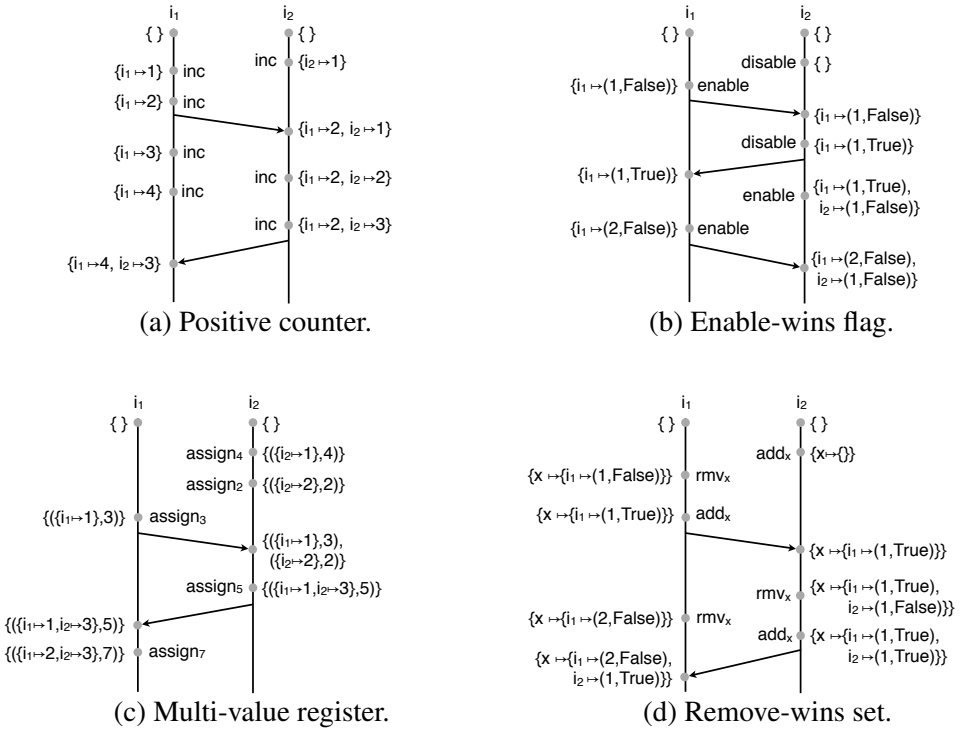
The definitions of join and the order come directly from the isomorphism. Upon a join, given two antichains, all elements that are concurrent are kept, but any element that is subsumed by a greater element is removed.

$$a \sqcup b = \mathsf{maximal}(a \cup b)$$

$$a \sqsubseteq b = \forall x \in a \cdot \exists y \in b \cdot x \sqsubseteq y$$

$$\perp = \{\}$$

In [7] a similar structure, using vector clocks to capture poset ordering, is described as a ldom lattice and referred to as a *dominating set*.



(a) Positive counter. (b) Enable-wins flag.

(c) Multi-value register. (d) Remove-wins set.

Figure 1: Example executions.

The maximal elements construction is the basis for the creation of *multi-value registers* that can store both single values and multiple values, when concurrent assignment occurs (see Section 5.7). This is the core data-type for tracking updates to shopping carts in the original Amazon Dynamo framework [10], and occurs in derived implementations such as the Riak Key-Value Store [4].

5 Abridged Catalog

In order to exemplify the composition constructs we present a small set of example CRDTs. Simple query functions are included and all mutators are inflations. Notice that join does not need to be defined as it follows from the composition rules that were introduced. Figure 1 shows example executions of most CRDTs discussed in this section.

5.1 Positive Counter

This simple form of counter can only increase. Replica nodes must have access to unique ids among a set I; each can only increment its position in a map of

ids to integers. While increment mutators are parametrized by id i the query is anonymous and simply inspects the state.

$$\mathsf{PCounter} = I \hookrightarrow \mathbb{N}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(a) \\ \mathsf{value}(a) &= \sum\{v \mid (i, v) \in a\} \end{aligned}$$

Notice that if a given node does not yet have an entry in the map and increments, then succ applies over $\perp$, which for $\mathbb{N}$ was defined to be 0.

5.2 Positive and Negative Counter

This variation allows for both increments and decrements. A solution is to pair two positive counters and consider the right side as negative. We use the standard functions fst() and snd() to respectively access the left and right elements of a pair.

$$\mathsf{PNCounter} = (I \hookrightarrow \mathbb{N}) \times (I \hookrightarrow \mathbb{N})$$

$$\begin{aligned} \mathsf{inc}_i(a) &= (\mathsf{apply}_i(\mathsf{succ})(\mathsf{fst}(a)), \mathsf{snd}(a)) \\ \mathsf{dec}_i(a) &= (\mathsf{fst}(a), \mathsf{apply}_i(\mathsf{succ})(\mathsf{snd}(a))) \\ \mathsf{value}(a) &= \sum\{v \mid (i, v) \in \mathsf{fst}(a)\} - \sum\{v \mid (i, v) \in \mathsf{snd}(a)\} \end{aligned}$$

An alternative way to obtain a similar result is to use a lexicographic pair and have the first element incremented when one needs to update the count on the second element.

$$\mathsf{LexCounter} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{Z}$$

$$\begin{aligned} \mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{id} \boxtimes \mathsf{succ})(a) \\ \mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \mathsf{pred})(a) \\ \mathsf{value}(a) &= \sum\{\mathsf{snd}(v) \mid (i, v) \in a\} \end{aligned}$$

$$\mathsf{pred}(x) = x - 1$$

While the PNCounter was one of the first CRDTs to be added to a production database, in Riak 1.4 [4], the competing Cassandra database had its own counter implementations based on the LWW strategy. Interestingly it proved to be difficult to avoid semantic anomalies in the behaviour of those early counters, and since Cassandra 2.1, a new counter was introduced [8] in line with the LexCounter.

5.3 Enable-wins Flag

A boolean flag that can be flipped, implemented in Riak under the name flag data type [3]. It uses a lexicographic pair per replica, where the left (more significant) element is a grow-only counter and the right element is a boolean. Enabling the flag increases the counter and resets the flag to False, for the replica entry; disabling the flag sets all booleans to True (maintaining the counters). The fact that only the enable operation increases the counter, ensures that this operation takes precedence over the disable operation. Flag starts disabled.

$$\mathsf{EWFlag} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{enable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\ \mathsf{disable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \end{aligned}$$

$$\mathsf{value}(a) = \exists i, n \cdot (i, (n, \mathsf{False})) \in a$$

5.4 Disable-wins Flag

The disable-wins flag is a dual construction of the enable-wins flag, and uses the same state lattice. Disabling the flag increases the counter, while enabling the flag sets all boolean to True. Flag starts enabled.

$$\mathsf{DWFlag} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned} \mathsf{disable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\ \mathsf{enable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \end{aligned}$$

$$\mathsf{value}(a) = \nexists\, i, n \cdot (i, (n, \mathsf{False})) \in a$$

5.5 Add-wins Set

A set with add-wins semantics can be derived by creating unique tokens whenever a new element is inserted, using for that a grow-only counter per replica, and canceling these tokens, by setting a boolean to True, upon removal. Only elements supported by non-canceled tokens are considered to be in the set.

$$\mathsf{AWSet} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$
\begin{aligned}
\mathsf{add}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{rmv}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a)
\end{aligned}
$$

$$
\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \exists i, n \cdot (i, (n, \mathsf{False})) \in m
$$

5.6 Remove-wins Set

A set with remove-wins semantics is derived by a dual construction to the previous one, while sharing the same state lattice. Removal creates unique tokens, and additions need to cancel all remove tokens that are visible in the state.

$$
\mathsf{RWSet} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}
$$

$$
\begin{aligned}
\mathsf{rmv}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{add}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a)
\end{aligned}
$$

$$
\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \nexists i, n \cdot (i, (n, \mathsf{False})) \in m
$$

5.7 Multi-value Register

A non-optimized multi-value register can be derived by lexicographic coupling a version vector clock $I \hookrightarrow \mathbb{N}$ with a payload value V. When a new value v is to be assigned, a new clock, greater than all visible clocks in the state, is created and coupled with the value. These pairs are kept in an antichain of maximal elements. Thus, upon merge, concurrently assigned values will be collected, but any subsequent assignment will again reduce the state to a single pair.

$$
\mathsf{MVRegister} = \mathcal{M}((I \hookrightarrow \mathbb{N}) \boxtimes V)
$$

$$
\begin{aligned}
\mathsf{assign}_{v,i}(a) &= \{\mathsf{apply}_i(\mathsf{succ})(\bigsqcup \{c \mid (c, v') \in a\}) \boxtimes v\} \\
\mathsf{values}(a) &= \{v \mid (c, v) \in a\}
\end{aligned}
$$

Notice that the value is never updated without creating a new clock. Thus, lexicographic comparison (needed for the operation of the maximals join) is always

decided by the first component, and V can be any opaque payload with no need for a partial order.

It is possible to improve the multi-value register construction, by keeping single tags with each value entry and storing a common causal context [26], and by compacting concurrent assignment of identical values [6]. Making such an improvement, while automatically deriving the join by lattice composition is still an open problem.

6 Closing Remarks

This report collects several composition techniques for lattices, adopts the notion of inflation and shows how it applies to the specification of state based CRDTs over lattices. Most of the lattice compositions are very standard techniques from order theory [9]. An early version of this work was presented at Schloss Dagstuhl under the title *Composition of Lattices and CRDTs* and the summary of the presentation is available at [12]. Most of the CRDT constructions used here are influenced by work in [2, 5–7, 20, 21].

The CRDTs selected for this small abridged catalog illustrate the potential of lattice composition, but do not cover the whole spectrum of known CRDTs, neither aims to be optimized. Further analysis on efficient implementations and optimality results can be found on [1, 6].

Acknowledgements. The work presented was partially supported by FCT-MCTES-PT NOVA LINCS project (UID/CEC/04516/2013), EU FP7 SyncFree project (609551), EU H2020 LightKone project (732505), and SMILES line in project TEC4Growth (NORTE-01-0145-FEDER-000020).

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 2017. http://dx.doi.org/10.1016/j.jpdc.2017.08.003.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference, (DAIS'14)*, pages 126–140. Springer, 2014. http://dx.doi.org/10.1007/978-3-662-43352-2_11.

[3] Basho. Riak datatypes, Retrieved 22-Dec-2015. http://github.com/basho.

[4] Basho. Riak 1.4, Retrieved 4-Jan-2016. https://github.com/basho/riak/blob/1.4/RELEASE-NOTES.md.

[5] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *Distributed Computing: 26th International Symposium (DISC'12)*, volume 7611 of *LNCS*, pages 441–442. Springer, 2012. http://dx.doi.org/10.1007/978-3-642-33651-5_48.

[6] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, pages 271–284. ACM, 2014. http://doi.acm.org/10.1145/2535838.2535848.

[7] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*, pages 1:1–1:14. ACM, 2012. http://doi.acm.org/10.1145/2391229.2391230.

[8] Datastax. What's New in Cassandra 2.1: Better Implementation of Counters, Retrieved 4-Jan-2016. http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters.

[9] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 205–220. ACM, 2007. http://doi.acm.org/10.1145/1294261.1294281.

[11] Jonathan Ellis. Why cassandra doesn't need vector clocks, Datastax, Retrieved 4-Jul-2017. https://www.datastax.com/dev/blog/why-cassandra-doesnt-need-vector-clocks.

[12] Bettina Kemme, André Schiper, G. Ramalingam, and Marc Shapiro. Dagstuhl seminar review: Consistency in distributed systems. *SIGACT News*, 45(1):67–89, 2014. http://doi.acm.org/10.1145/2596583.2596601.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. http://doi.acm.org/10.1145/359545.359563.

[14] Michael Owen. Using Erlang, Riak and the ORSWOT CRDT at bet365 for scalability and performance, Retrieved 17-Jul-2017. http://www.erlang-factory.com/euc2015/michael-owen.

[15] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983. http://dx.doi.org/10.1109/TSE.1983.236733.

[16] Peter Bourgon. Consistency without Consensus: CRDTs in Production at SoundCloud, Retrieved 22-Dec-2015. http://www.infoq.com/presentations/crdt-soundcloud.

[17] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference (USTC'94)*, pages 12–12. USENIX Association, 1994. http://dl.acm.org/citation.cfm?id=1267257.1267269.

[18] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990. http://dx.doi.org/10.1109/12.54838.

[19] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib. Comput.*, 7(3):149–174, March 1994. https://doi.org/10.1007/BF02277859.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011. http://hal.archives-ouvertes.fr/inria-00555588/.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, volume 6976 of *LNCS*, pages 386–400. Springer, 2011. http://dx.doi.org/10.1007/978-3-642-24550-3_29.

[22] Douglas B. Terry, Marvin M. Theimer, Karin Peterson, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Mobility*, pages 322–334. ACM, 1999. http://dl.acm.org/citation.cfm?id=303461.342780.

[23] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979. http://doi.acm.org/10.1145/320071.320076.

[24] Todd Hoff. How League of Legends Scaled Chat to 70 Million Players - It takes a lot of Minions, Retrieved 22-Dec-2015. http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.

[25] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008. http://doi.acm.org/10.1145/1466443.1466448.

[26] Marek Zawirski, Carlos Baquero, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Eventually consistent register revisited. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC'16)*, pages 9:1–9:3. ACM, 2016. http://doi.acm.org/10.1145/2911151.2911157.