# Coordination-free Byzantine Replication with Minimal Communication Costs

**Jelle Hellings**
Exploratory Systems Lab, Department of Computer Science, University of California, Davis
Davis, CA 95616-8562, USA
jhellings@ucdavis.edu

**Mohammad Sadoghi**
Exploratory Systems Lab, Department of Computer Science, University of California, Davis
Davis, CA 95616-8562, USA
msadoghi@ucdavis.edu

## Abstract

State-of-the-art fault-tolerant and federated data management systems rely on fully-replicated designs in which all participants have equivalent roles. Consequently, these systems have only limited scalability and are ill-suited for high-performance data management. As an alternative, we propose a hierarchical design in which a *Byzantine cluster* manages data, while an arbitrary number of *learners* can reliable learn these updates and use the corresponding data.

To realize our design, we propose the *delayed-replication algorithm*, an efficient solution to the *Byzantine learner problem* that is central to our design. The delayed-replication algorithm is coordination-free, scalable, and has minimal communication cost for all participants involved. In doing so, the delayed-broadcast algorithm opens the door to new high-performance fault-tolerant and federated data management systems. To illustrate this, we show that the delayed-replication algorithm is not only useful to support specialized learners, but can also be used to reduce the overall communication cost of permissioned blockchains and to improve their storage scalability.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Information systems → Distributed database transactions

**Keywords and phrases** Byzantine learner, coordination-free checkpoint protocol, delayed-replication, information dispersal, consensus

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2020.22

## 1 Introduction

Recently saw the introduction of several blockchain-inspired database systems and blockchain fabrics [5, 6, 24, 25, 54, 55]. At the same time, there is also a huge interest from public and private sectors in blockchain technology (e.g., [7, 11, 13, 16, 18, 27, 29, 34, 35, 40, 41, 52, 55, 61, 62, 65, 73]). In each of these systems and use cases, blockchain technology is used to provide *fault-tolerant and federated data management*: systems in which independent participants (e.g., different companies) together manage a single common database and that continuously provide reliable services even when some of the participants are compromised. The interest in fault-tolerant and federated data management is easy explained by the huge societal and economic impact of recent cyberattacks on data-based services [31, 56, 57, 58, 68], and on the huge negative economic impact of bad data [23, 39, 64].

Blockchain techniques build upon traditional distributed consensus [38, 53]: both traditional techniques and their blockchain counterparts provide fault-tolerant and federated data management via a *fully-replicated* design in which all participants (replicas) maintain a full copy of all data and participate in modifying this data. To do so, traditional consensus—which are also used in *permissioned blockchains* in which the identities of all participants are known—requires vast amounts of communication between all participants (e.g., [8, 9, 14, 15, 45, 46, 59, 69, 70]). Consequently, systems using traditional consensus have difficulty scaling up to hundreds of participants. Techniques used in anonymous permissionless blockchains such as Bitcoin can effectively support thousands of participants, however. Unfortunately, these blockchain techniques incur massive *computational costs* on all participants, which has raised questions about the sustainability of the *energy consumption* of these systems [21, 72]. Even with these computational costs, the performance of Bitcoin is abysmal, as Bitcoin can only process 7 transactions per second [61].

We see the necessity of fault-tolerant and federated data management but—as outlined above—we also believe that the current state-of-the-art techniques are too limited and lack scalability. To combat these limitations, we believe there is a strong need for the development of more refined scalable designs that provide fault-tolerant and federated data management.

## 1.1 Our Vision: Specializing for Read-only Workloads

In many practical distributed database and data processing systems, a distinction is made between read-only workloads and update-workloads [1, 19, 20, 22, 32, 60, 70]. Typically, read-only workloads are isolated to a single replica, whereas update-workloads are executed by all replicas (e.g., via a commit protocol [30, 36, 67]). In most cases this improves scalability significantly, as the majority of workloads are read-only and can be processed in parallel by individual replicas. Unfortunately, such read-only single-replica optimizations cannot be applied to state-of-the-art fault-tolerant and federated data management: fault-tolerant systems need to assure validity of the result of every read-only query in the presence of malicious replicas. These systems do so by executing every query at all replicas, after which the issuer of the query can compare the query outcomes and determine which outcome is valid (supported by a majority).

In many practical situations, workloads need access to the full history of all the data managed or to large portions thereof. Examples of such workloads are analytics, data provenance, machine learning, and data visualization. For data-hungry workloads, it makes little sense to retrieve all data in an inefficient way via read-only queries. Furthermore, these workloads are typically computational complex, ruling out their integration within a fault-tolerant system. To enable these practical workloads, we propose an alternative *hierarchical design*. This hierarchical design is sketched in Figure 1.

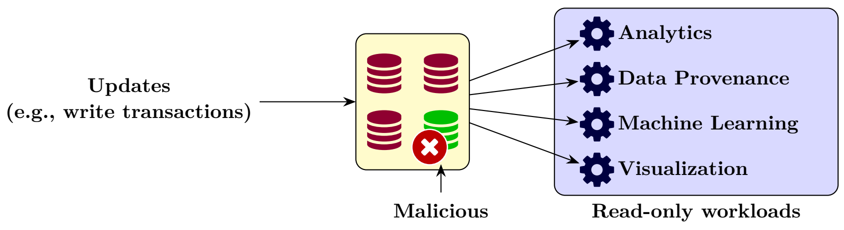

**Figure 1** Schematic overview of *hierarchical* fault-tolerant and federated data management. At the core is a *Byzantine cluster* that manages and stores all data in a fault-tolerant manner. Some of the replicas in this core can crash or be malicious. The managed data is used by many *independent read-only participants*, e.g., for analytics, data provenance, machine learning, and visualization. To do so, these participants do not need to partake in managing and storing the data, they only need to *reliably learn the data*.

In our design, we propose that a *Byzantine cluster* of replicas (e.g., a permissioned blockchain system) manages the data by coordinating data updates. As the cluster is Byzantine fault-tolerant, it can be used to provide fault-tolerant and federated data management.

Dedicated *learners*, independent of the Byzantine cluster, can register themselves at the Byzantine cluster to receive all data updates. These learners will receive the stream of data updates made in the cluster and will receive these updates in an efficient and reliable manner. On these learners, data-hungry and compute intensive read-only workloads (e.g., analytics, data provenance, machine learning, and data visualization) can be performed efficiently without affecting the Byzantine cluster. Learners can also be deployed in trusted environments close to end-users and act as read-only proxies. In this read-only proxy capacity, learners provide end-users with high performance, low latency, read-only access to the data. In this hierarchical design, learners cannot directly modify the data, but can still forward data update requests to the Byzantine cluster. The Byzantine cluster can, in turn, assure reliable processing of such updates.

### 1.2 The Need for the Byzantine Learner Problem

To enable the hierarchical design proposed in the previous section, we need to develop techniques to reliably sent the data updates made by a *Byzantine cluster* to independent *learners*—which we refer to as the *Byzantine learner problem.* In specific, our contributions are as follows:

1. We formalize the Byzantine learner problem.
2. We demonstrate that the straightforward *pull-based* solution to this learning problem is highly inefficient and enables several attacks.
3. To address the Byzantine learner problem, we propose the *delayed-replication algorithm*, a coordination-free, push-based, scalable algorithm with minimal communication cost for both the sending cluster and the receiving learner.
4. We provide three specialized variants of the delayed-replication algorithm, whose characteristics are summarized in Figure 2. The basic variant does not use checksums and can deal with clusters in which replicas can *crash.* We also provide a variant that uses *simple checksums* to deal with *Byzantine replicas* that sent corrupted or otherwise invalid messages. The final variant uses *tree checksums* to aid learners in discarding corrupted or otherwise invalid messages with low computational costs. These tree checksums only add minimal communication costs for all participants involved.
5. To further underline the strengths of the delayed-replication algorithm, we show that the delayed-replication algorithm can be used to improve the design of permissioned blockchain systems and other types of Byzantine clusters. First, we show how delayed-replication techniques enables scalable shared storage designs for permissioned blockchain systems, allowing them to turn away from wasteful state-of-the-art fully-replicated designs. Then, we show how the delayed-replication algorithm can be used within permissioned blockchain systems to reduce the communication complexity of coordinating data updates.

## 2 Formalizing the Byzantine Learner Problem

We model a *system* as a tuple $(\mathfrak{R}, \mathfrak{L})$, in which $\mathfrak{R}$ is a *Byzantine cluster* of replicas that make *update decisions* and $\mathfrak{L}$ is a set of *learners* that want to learn these *update decisions.* We assign each replica $\textsc{r} \in \mathfrak{R}$ a unique identifier $\mathsf{id}(\textsc{r})$ with $0 \leq \mathsf{id}(\textsc{r}) < |\mathfrak{R}|$. We write $\mathcal{B} \subseteq \mathfrak{R}$ to denote the set of *Byzantine replicas* that can behave in arbitrary, possibly coordinated and malicious, manners; we write $\mathcal{C} \subseteq \mathfrak{R}$ to denote the set of *crashed replicas* that behave correctly up till some point after which they stop participating; and we write $\mathcal{G} = \mathfrak{R} \setminus (\mathcal{B} \cup \mathcal{C})$ to denote the set of *non-faulty (good) replicas* in $\mathfrak{R}$. We assume that non-Byzantine replicas

| System | Checksum | Complexity for the learner | | |
|---|---|---|---|---|
| | | *Data sent per replica* | *Data received* | *Decode steps* |
| $\mathbf{b} = 0$ | None | $\mathcal{O}(s/\mathbf{g})$ | $\mathcal{O}(s(\mathbf{n}/\mathbf{g}))$ | $u/\mathbf{n}$ |
| $\mathbf{b} < \mathbf{g}$ | Simple | $\mathcal{O}(s/\mathbf{g})$ | $\mathcal{O}(s(\mathbf{n}/\mathbf{g}))$ | $\binom{\mathbf{g}+\mathbf{b}}{\mathbf{g}}(u/\mathbf{n})$ |
| $\mathbf{b} < \mathbf{g}$ | Tree | $\mathcal{O}(s/\mathbf{g} + (u/\mathbf{n})\log(\mathbf{n}))$ | $\mathcal{O}(s(\mathbf{n}/\mathbf{g}) + u\log(\mathbf{n}))$ | $u/\mathbf{n}$ |

**Figure 2** Overview of *delayed-replication algorithms* running on a cluster of $\mathbf{n}$ replicas, of which $\mathbf{b}$ are Byzantine and $\mathbf{g}$ are non-faulty. The first two columns describe the system conditions and the checksums used. The last three columns provide the complexity to sent a journal with $u$ updates and storage size $s$ to a learner in terms of the data sent per replica, data received by the learner, and the worst-case number of decode steps the learner needs to perform.

behave in accordance to the algorithms and are deterministic: on identical inputs, non-faulty replicas must produce identical outputs. Notice that we do not make any assumptions on the learners, each learner can be malicious without affecting the operations in $\mathfrak{R}$. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{b} = |\mathcal{B}|$, $\mathbf{c} = |\mathcal{C}|$, and $\mathbf{g} = |\mathcal{G}|$ to denote the number of replicas, Byzantine replicas, crashed replicas, and non-faulty replicas, respectively. Finally, we assume that $\mathbf{g} > \mathbf{b}$, a minimal condition to distinguish Byzantine and non-Byzantine behavior.

▶ **Definition 2.1.** *Let* $(\mathfrak{R}, \mathfrak{L})$ *be a system. The* Byzantine learner problem *states that each learner in* $\mathfrak{L}$ *will eventually learn of the update decisions made by* $\mathfrak{R}$*.*

We will formalize the Byzantine learner problem in terms of learning *journal updates*. Let $(\mathfrak{R}, \mathfrak{L})$ be a system. We assume that each replica $\textsc{r} \in \mathfrak{R}$ maintains an append-only *update journal* $\mathbb{J}_{\textsc{r}}$ that consists of a sequence of *data updates* (e.g., write transactions in a database system). To work with sequences, we introduce the following notations. Let $S = [s_0, \dots, s_{m-1}]$ be a sequence. We write $S[i]$ to denote $s_i$, $S[i:j]$ to denote $[s_i, \dots, s_{j-1}]$, and $|S|$ to denote the length $m$ of $S$. Finally, if $T$ is also a sequence, then $S$ is a *prefix* of $T$, denoted $S \preceq T$, if $|S| \leq |T|$ and $S = T[0:|S|]$. We refer to any subsequence $S[i : i + \mathbf{n}]$, $i \bmod \mathbf{n} = 0$, as a *block*.

We assume that the non-Byzantine replicas all make *the same update decisions* in the same order (e.g., by utilizing a consensus protocol such as `Paxos` or `Pbft` [14, 15, 46, 47]). These updates are not necessarily registered at each replica at exactly the same time. Consequently, we can only assume that, for each $\textsc{r}, \textsc{q} \in (\mathcal{G} \cup \mathcal{C})$, either $\mathbb{J}_{\textsc{r}} \preceq \mathbb{J}_{\textsc{q}}$ or $\mathbb{J}_{\textsc{q}} \preceq \mathbb{J}_{\textsc{r}}$. We write $\mathbb{J}_{\mathfrak{R}}$ to denote the unique journal $\mathbb{J}_{\textsc{q}}$, $\textsc{q} \in \mathcal{G}$, that contains the maximum-length sequence of update decisions all non-faulty replicas agree on. Hence, $\mathbb{J}_{\mathfrak{R}} \preceq \mathbb{J}_{\textsc{r}}$ for all $\textsc{r} \in \mathcal{G}$.

▶ **Example 2.2.** Consider a Byzantine cluster $\mathfrak{R} = \{\textsc{r}_0, \textsc{r}_1, \textsc{r}_2, \textsc{b}\}$ with

$$\mathbb{J}_{\textsc{r}_0} = [u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7]; \qquad \mathbb{J}_{\textsc{r}_1} = [u_0, u_1, u_2, u_3, u_4, u_5, u_6];$$
$$\mathbb{J}_{\textsc{r}_2} = [u_0, u_1, u_2, u_3, u_4, u_5, u_6]; \qquad \mathbb{J}_{\textsc{b}} = [u_0, u_1, u_2, u_3', u_4'].$$

The update journal of replica $\textsc{b}$ diverges from the other replicas and, hence, $\textsc{b}$ must be Byzantine. The three non-faulty replicas share the update journal $\mathbb{J}_{\mathfrak{R}} = [u_1, u_2, u_3, u_4, u_5, u_6]$. Currently, the cluster is deciding on the eight update $u_7$. This update is already fully processed by $\textsc{r}_0$, whereas replicas $\textsc{r}_1$ and $\textsc{r}_2$ are still processing this update.

▶ **Definition 2.3.** *Let* $(\mathfrak{R}, \mathfrak{L})$ *be a system and* $\textsc{l} \in \mathfrak{L}$ *a learner. For every* $i$*,* $0 \leq i < |\mathbb{J}_{\mathfrak{R}}|$*, the* Byzantine learner problem *states that* $\textsc{l}$ *will eventually learn of the* $i$*-th update decision* $\mathbb{J}_{\mathfrak{R}}[i]$*. At the same time, no Byzantine replica* $\textsc{b} \in \mathcal{B}$ *can convince* $\textsc{l}$ *that any other update was the* $i$*-th update decision made.*

Notice that we only specified the data model of replicas. We did not specify how the learners store data or process data, we only specified that the learners will receive *all* update decisions made by the replicas. Indeed, the specifics of what a learner does with the updates received depend on the workload for which the learner is designed.

▶ **Example 2.4.** Consider the Byzantine cluster $\mathfrak{R}$ from Example 2.2. A learner $\textsc{l}$ will be able to learn the updates $u_0$, $u_1$, $u_2$, $u_3$, $u_4$, $u_5$, and $u_6$. The learner will not yet be able to learn $u_7$, as this update is still being processed by some non-faulty replicas in $\mathfrak{R}$. Replica $\textsc{b}$ will never be able to convince the learner that the updates $u_3'$ or $u_4'$ happened, as $\textsc{b}$ is Byzantine. The learner $\textsc{l}$ can store the updates it learned in a temporal database view that provides access to historical data, e.g., for in-depth analysis.

As a simple solution to the *Byzantine learner problem*, consider a system in which each replica can be queried for their journal content. Any learner $\textsc{l}$ can now determine the $i$-th journal update by simply querying different replicas in $\mathfrak{R}$. As soon as $\textsc{l}$ receives $\mathbf{b}+1$ identical responses, it is ensured that at least one of these responses came from a non-Byzantine replica and, hence, must be the valid $i$-th journal update. Unfortunately, this simple and naive solution has several major weaknesses that can be exploited by malicious participants:

▶ **Example 2.5.** Firstly, there is the issue of *load balancing* due to a lack of coordination. As all learners have to query for journal updates independently, they can all end up querying the same non-faulty replica $\textsc{r} \in \mathcal{G}$. Due to the amount of queries, $\textsc{r}$ has to dedicate most of its resources to answering these queries. Consequently, $\textsc{r}$ will have fewer resources available for its other tasks, e.g., for deciding on new data updates. In the worst case, this can reduce the data update throughput of $\mathfrak{R}$. Secondly, the issue of load balancing can be *exploited* by the ability of *malicious learners* to coordinately target some non-faulty replicas, which could overload these replicas in an attempt to impede the services of $\mathfrak{R}$.

Moreover, as learners do not have a reliable way to distinguish between non-faulty replicas that are slow, crashed replicas, and Byzantine replicas, they have to always query at least $\mathbf{b}+\mathbf{c}+1$ distinct replicas in $\mathfrak{R}$ to have a guarantee on an outcome (as $\mathbf{b}+\mathbf{c}$ replicas could be Byzantine or have crashed and consequently not respond). Furthermore, an additional $\mathbf{b}$ distinct replicas in $\mathfrak{R}$ need to be queried to assure that the majority of all received outcomes come from non-Byzantine replicas (as $\mathbf{b}$ replicas could be Byzantine and respond with invalid or corrupted outcomes). This makes learning an update *unnecessary expensive*.

In Section 3, we propose the delayed-replication algorithm to provide reliable high-performance Byzantine learning that does not suffer from the shortcomings of the above naive simple solution.

In the following, we assume *asynchronous reliable communication*: all messages send by non-faulty replicas will eventually arrive at their destination. We also assume *authenticated communication*: on receipt of a message $m$ from replica $\textsc{r} \in \mathfrak{R}$, one can determine that $\textsc{r}$ did sent $m$ if $\textsc{r} \notin \mathcal{B}$; and one can only determine that $m$ was sent by a replica in $\mathcal{C} \cup \mathcal{G}$ if $\textsc{r} \in (\mathcal{C} \cup \mathcal{G})$. Hence, Byzantine replicas are able to impersonate each other, but are not able to impersonate non-Byzantine replicas. Authenticated communication is a minimum requirement to deal with Byzantine behavior and can be implemented using message authentication codes [43, 50].

## 3 The Delayed-Replication Algorithm

Next, we propose the *delayed-replication algorithm*, which provides an efficient solution to the Byzantine learner problem. Our delayed-replication algorithm uses *information dispersal* [63] to balance the load among all non-faulty replicas and to minimize overall communication costs. The delayed-replication algorithm itself consists of two parts: the *information dispersal step*, which is executed by the replicas in $\mathfrak{R}$, and the *information learning step*, which is executed by the learners in $\mathfrak{L}$.

## 3.1 Information Dispersal

We use an *information dispersal algorithm* that is able to *encode* any value $v$ with storage size $\|v\|$ into $\mathbf{n}$ pieces $v_i$, $0 \leq i < \mathbf{n}$, such that $v$ can be *decoded* from every set of $\mathbf{g}$ distinct pieces. We assume that the information dispersal algorithm is *optimal* in the sense that each piece $v_i$ has size $\|v_i\| \leq \lceil \|v\|/\mathbf{g} \rceil$. Hence, the minimal number of pieces necessary for recovering $v$ by decoding, $\mathbf{g}$ pieces, have a combined storage size of $\mathbf{g} \lceil \|v\|/\mathbf{g} \rceil \approx \|v\|$. The information dispersal algorithm (IDA) of Rabin provides these properties [63].

We assume that each non-Byzantine replica $\mathrm{R} \in (\mathcal{C} \cup \mathcal{G})$ is equipped with IDA. We write $\texttt{slice}_{\mathrm{R}}(v)$, for any value $v$, to denote the $\mathsf{id}(\mathrm{R})$-th piece $v_{\mathsf{id}(\mathrm{R})}$ obtained by encoding $v$. With these assumptions and notations, we have $\|\texttt{slice}_{\mathrm{R}}(v)\| \leq \lceil \|v\|/\mathbf{g} \rceil$.

▶ **Example 3.1.** Consider a cluster $\mathfrak{R} = \{\mathrm{R}_0, \mathrm{R}_1, \mathrm{R}_2, \mathrm{B}\}$ with $\mathcal{B} = \{\mathrm{B}\}$ and $\mathcal{C} = \emptyset$. Hence, $\mathbf{g} = 3$ and $\mathbf{b} = 1$. Let $v$ be a piece of data. When using the encode step of IDA, we obtain pieces $v_0, v_1, v_2, v_3$ with $\|v_0\| = \|v_1\| = \|v_2\| = \|v_3\| = \lceil \|v\|/3 \rceil$. Consequently, $\texttt{slice}_{\mathrm{R}_0}(v) = v_0$, $\texttt{slice}_{\mathrm{R}_1}(v) = v_1$, $\texttt{slice}_{\mathrm{R}_2}(v) = v_2$, and, finally, $\texttt{slice}_{\mathrm{B}}(v) = v_3$.

Now consider any learner $\mathrm{L} \in \mathfrak{L}$. Upon obtaining any three valid and distinct pieces, L can use the decode step of IDA to reconstruct $v$. As the replicas $\mathrm{R}_0$, $\mathrm{R}_1$, and $\mathrm{R}_2$ are non-faulty, L will always be able to obtain $v_0$, $v_1$, and $v_2$. Hence, L can reconstruct $v$. We notice that $\|v_0\| + \|v_1\| + \|v_2\| = 3 \lceil \|v\|/3 \rceil \approx \|v\|$. Hence, the communication required for L to reconstruct $v$ is minimal (due to IDA being optimal).

## 3.2 The Information Dispersal Step

In the *information dispersal step*, every replica $\mathrm{R} \in \mathfrak{R}$ is instructed to broadcast the update decisions appended to their journal after every block $\mathbb{B}$ of $\mathbf{n}$ appends. The pseudo-code for the information dispersal step can be found in Figure 3. To minimize communication costs, replicas will encode the block $\mathbb{B}$ using an optimal information dispersal algorithm (Line 4). To allow learners to validate the correctness of the encoded block, replicas will include a *checksum* of $\mathbb{B}$. The exact type of checksum used depends on the type of attacks the delayed-replication algorithm needs to be able to deal with, and we refer to the information learning step for details on the types of checksums supported (Section 3.3 and Section 3.4). After encoding, each replica broadcasts the encoded block and the checksum to all learners (Line 5). In doing so, the information dispersal step provides reliable replication of sufficient information among *all* learners such that each learner can reconstruct any segment of $\mathbf{n}$ update decisions. We refer to Figure 4 for a schematic representation of the interactions between replicas and a learner due to the information dispersal step.

We notice that the information dispersal step is a *push-based* algorithm that pushes the update journal to all learners without any coordination. Additionally, the total communication cost of the information dispersal step is shared equally among all participating replicas, independent of the behavior of any faulty replicas. This is in sharp contrast with the simple and naive *pull-based* approach of Section 2, which is at the basis of many *practical checkpoint algorithms* (see, e.g., Section 4.2). Next, we show that the communication complexity of the information dispersal step is low.

```
1: event R appends a new decision to $\mathbb{J}_R$ do
2:     if $\mathbb{J}_R \neq [\,]$ and $|\mathbb{J}_R| \bmod \mathbf{n} = 0$ then
3:         $\mathbb{B} := \mathbb{J}_R[|\mathbb{J}_R| - \mathbf{n} : |\mathbb{J}_R|]$.
4:         $s, c := \texttt{slice}_R(\mathbb{B}), \texttt{checksum}(\mathbb{B})$.
5:         Broadcast $(|\mathbb{J}_R|, s, c)$ to all learners $L \in \mathfrak{L}$.
```

**Figure 3** The *information dispersal step* of the delayed-replication algorithm running at every non-faulty replica $R \in \mathcal{G}$.

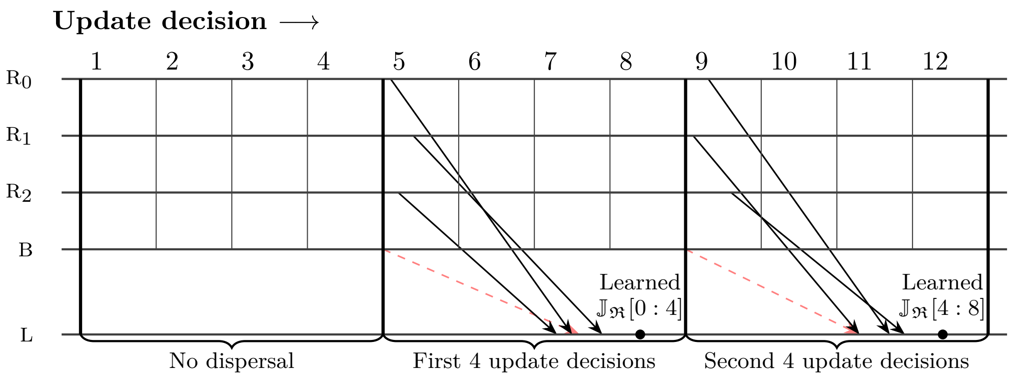
Update decision ⟶
1
2
3
4
5
6
7
8
9
10
11
12
R0
R1
R2
B
L
Learned $\mathbb{J}_{\mathfrak{R}}[0:4]$
Learned $\mathbb{J}_{\mathfrak{R}}[4:8]$
No dispersal
First 4 update decisions
Second 4 update decisions

**Figure 4** A schematic representation of the interactions between a cluster $\mathfrak{R} = \{R_0, R_1, R_2, B\}$ and a learner L participating in the information dispersal step. The replica B is Byzantine and sends invalid messages. The other replicas repeatedly send a valid message encoding 4 decisions from their update journal. After receiving these messages, L is able to reconstruct (learn) the update decisions made by $\mathfrak{R}$. In specific, after the $\mathbf{n} = 4$-th update decision, L will start receiving messages from which it can reconstruct the first four update decisions.

▶ **Theorem 3.2.** *Consider the information dispersal step of Figure 3 running at replica $R \in \mathcal{G}$ after R appends the $\rho$-th decision, $\rho \geq 1$, to $\mathbb{J}_R$. After this step, R has sent $\lfloor \rho/\mathbf{n} \rfloor$ messages to each learner with a total size of $\mathcal{O}(c(\rho/\mathbf{n}) + \|\mathbb{J}_R\|/\mathbf{g})$, in which $c$ is the size of a checksum.*

**Proof.** Notice that R only broadcasts messages after every $i$-th decision, $i \geq 1$ and $i \bmod \mathbf{n} = 0$. Hence, after the $\rho$-th decision, R will have broadcasted $m = \lfloor \rho/\mathbf{n} \rfloor$ messages. Consider the messages sent by R to any learner $L \in \mathfrak{L}$. In these messages, the pieces $\texttt{slice}_R(\mathbb{J}_R[(i-1)\mathbf{n} : i\mathbf{n}])$, $1 \leq i \leq m$, have a non-constant size and we assume that the remainder of each message has size $\gamma = \Theta(c)$. Hence, in total, the $m$ messages send to L have size $\sigma$ at most

$$\sigma \leq \sum_{1 \leq i \leq m} (\gamma + \|\texttt{slice}_R(\mathbb{J}_R[(i-1)\mathbf{n} : i\mathbf{n}])\|)$$
$$\leq \gamma m + \sum_{1 \leq i \leq m} \left\lceil \frac{\|\mathbb{J}_R[(i-1)\mathbf{n} : i\mathbf{n}]\|}{\mathbf{g}} \right\rceil \leq \gamma m + \sum_{1 \leq i \leq m} \left(1 + \frac{\|\mathbb{J}_R[(i-1)\mathbf{n} : i\mathbf{n}]\|}{\mathbf{g}}\right)$$
$$\leq \gamma m + m + \frac{\|\mathbb{J}_R[0 : \mathbf{n}m]\|}{\mathbf{g}} \leq m(\gamma + 1) + \frac{\|\mathbb{J}_R\|}{\mathbf{g}} = \mathcal{O}(c(\rho/\mathbf{n}) + \|\mathbb{J}_R\|/\mathbf{g}).$$ ◀

Based on Theorem 3.2, it is straightforward to determine the number and size of messages received by each learner.

▶ **Corollary 3.3.** *Consider the learner $L \in \mathfrak{L}$ after it has received all messages sent by the information dispersal steps following the $\rho$-th decision in $\mathbb{J}_{\mathfrak{R}}$, $\rho \geq 1$. The learner L has* *received at most $\rho$ messages with a total size of $\mathcal{O}(c\rho + \|\mathbb{J}_{\mathfrak{R}}\|(\mathbf{n}/\mathbf{g}))$, in which $c$ is the size of a checksum.*

To conclude, we notice that the information dispersal step we provide assumes a steady flow of update decisions. If update decisions are infrequent, then information dispersal can be delayed arbitrary. To deal with periods of inactivity, the system can always resort to filling the current block of $\mathbf{n}$ updates with `null`-values (although this will reduce communication efficiency).

## 3.3 The Information Learning Step with Simple Checksums

In the previous section, we presented the information dispersal step that will broadcast an encoded block of journal updates from $\mathbb{J}_{\mathfrak{R}}$ to each learner $\text{L} \in \mathfrak{L}$. In this section, we show how L can reliable reconstruct these journal updates from the encoded information. To provide resilience against Byzantine replicas, we will use *simple checksums* $\texttt{checksum}(\mathbb{B}) = \texttt{hash}(\mathbb{B})$, in which $\texttt{hash}(\cdot)$ is a *collision-resistant hash function* that maps an arbitrary value $v$ to a numeric value $\texttt{hash}(v)$ in a bounded range [43, 50]. We assume that it is practically impossible to find another value $v'$, $v \neq v'$, such that $\texttt{hash}(v) = \texttt{hash}(v')$. These simple checksums have a constant size independent of $\mathbf{n}$ or $\mathbf{g}$.

▶ **Theorem 3.4.** *Consider the learner $\text{L} \in \mathfrak{L}$ after it has received all messages sent by the information dispersal steps following the $\rho$-th decision in $\mathbb{J}_{\mathfrak{R}}$, $\rho \geq 1$. If $\mathbf{g} > \mathbf{b}$, then, after receiving these messages, L can reconstruct the first $\mathbf{n}\lfloor \rho/\mathbf{n} \rfloor$ update decisions made by $\mathfrak{R}$ using at most $\binom{\mathbf{g}+\mathbf{b}}{\mathbf{g}}\lfloor \rho/\mathbf{n} \rfloor$ information dispersal decode steps.*

**Proof.** Let $i = \mathbf{n}\lfloor \rho/\mathbf{n} \rfloor$ be the last round after which L received update decisions. We assume that L has already reconstructed the first $(i-1)\mathbf{n}$ update decisions, and we show how L can reconstruct the block $\mathbb{B}$ containing update decisions $(i-1)\mathbf{n}, \dots, i\mathbf{n}-1$, this independent of the behavior of the Byzantine replicas. To initiate reconstruction of $\mathbb{B}$, L will collect messages of the form $(i, s_j, c_j)$, with $s_j$ an encoded piece of $\mathbb{B}$ and $c_j$ a checksum, of distinct replicas $\text{R}_j$ with $\mathsf{id}(\text{R}_j) = j$. Eventually, L will receive $\mathbf{g}$ messages from replicas in $\mathcal{C} \cup \mathcal{G}$, as all $\mathbf{g}$ replicas in $\mathcal{G}$ will send messages to L. Using these messages, L can reconstruct $\mathbb{B}$ by decoding the pieces contained in the received messages.

Byzantine replicas are able to send corrupted messages, however, which complicates construction of $\mathbb{B}$ from the messages received. Learners do not a-priori know which replicas are Byzantine. Hence, learners need to verify whether any block reconstructed from $\mathbf{g}$ collected messages is equivalent to $\mathbb{B}$. The first step in this verification process is to determine the checksum $\texttt{hash}(\mathbb{B})$. Consider the first $z > \mathbf{b}$ messages received. We distinguish two cases:

1. At least $\mathbf{b}+1$ messages have identical checksum $c$. In this case, at least one such message must be sent by a non-faulty replica. Hence, we have $c = \texttt{hash}(\mathbb{B})$.
2. At most $\mathbf{b}$ messages have identical checksums. In this case, some of the messages received have been sent by Byzantine replicas. As L will eventually receive $\mathbf{g} > \mathbf{b}$ messages from non-faulty replicas, and all these messages will contain the same checksum $\texttt{hash}(\mathbb{B})$, we can wait until more messages are received to determine the checksum $\texttt{hash}(\mathbb{B})$.

After determining $\texttt{hash}(\mathbb{B})$, L can simply reconstruct $\mathbb{B}$ by trying to decode every combination of $\mathbf{g}$ received pieces, this until eventually a block $b$ is constructed with $\texttt{hash}(b) = \texttt{hash}(\mathbb{B})$. In the worst case, L will have to wait until it receives $\mathbf{g}+\mathbf{b}$ messages before it receives $\mathbf{g}$ uncorrupted messages and it will have to try to decode $\binom{\mathbf{g}+\mathbf{b}}{\mathbf{g}}$ combinations of $\mathbf{g}$ pieces before it finds $\mathbf{g}$ pieces sent by non-Byzantine replicas.

The only way for Byzantine replicas to subvert the learning step is by finding a value $w$, $w \neq \mathbb{B}$, with $\texttt{hash}(w) = \texttt{hash}(\mathbb{B})$. As we assumed that $\texttt{hash}(\cdot)$ is a *collision-resistant hash* *function*, it is exceedingly hard for the Byzantine replicas to find such a value $w$. Furthermore, as $\mathbf{g}$ pieces are used during decoding and $\mathbf{g} > \mathbf{b}$, the Byzantine replicas not only need to find $w$, but must also find a way to encode $w$ such that it can be reconstructed using pieces provided by one or more non-faulty replicas. Hence, assuming reasonable limits on the computational resources, the Byzantine replicas are unable to subvert the learning step.[1] ◀

Notice that if the system has no Byzantine replicas ($\mathbf{b} = 0$), then the checksums can be omitted entirely, as every set of $\mathbf{g}$ pieces will decode to the searched-for block of update decisions. This strongly reduces the computational costs for the learner. By combining Theorem 3.2 and Theorem 3.4, we obtain:

▶ **Corollary 3.5.** *Consider the learner* $\textsc{l} \in \mathfrak{L}$ *and replica* $\textsc{r} \in \mathcal{G}$*. If* $\mathbf{g} > \mathbf{b}$*, then the delayed-replication algorithm with simple checksums guarantees*

1. $\textsc{l}$ *will learn the update journal* $\mathbb{J}_{\mathfrak{R}}$*;*
2. $\textsc{l}$ *will receive at most* $|\mathbb{J}_{\mathfrak{R}}|$ *messages with a total size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|(\mathbf{n}/\mathbf{g}))$*;*
3. $\textsc{l}$ *will only need worst-case* $\binom{\mathbf{g}+\mathbf{b}}{\mathbf{g}}(|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n})$ *information dispersal decode steps; and*
4. $\textsc{r}$ *will sent at most* $|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n}$ *messages to* $\textsc{l}$ *with a total size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g})$*.*

To conclude, we notice that Byzantine replicas that send corrupted messages are easily detectable by the learners. After a learner $\textsc{l} \in \mathfrak{L}$ has decoded $\mathbf{g}$ pieces into a valid block $\mathbb{B}$, it can simply encode this block and determine the exact value of each encoded piece a replica should have sent to $\textsc{l}$. Hence, after trying to subvert a learning step of $\textsc{l}$, Byzantine replicas can be recognized and be eliminated from future considerations. When the set of Byzantine replicas is relatively stable over time, we can use this approach towards detecting Byzantine behavior at $\textsc{l}$ to prevent the worst-case upper bound on the number of information dispersal decode steps, $\binom{\mathbf{g}+\mathbf{b}}{\mathbf{g}}$, from happening repeatedly.

## 3.4 The Information Learning Step with Tree Checksums

In the previous section, we have shown how learners can reliably reconstruct $\mathbb{J}_{\mathfrak{R}}$ in the presence of Byzantine replicas. In theory, this provided approach has a high computational overhead for the learners due to the worst-case combinatorics involved. In this section, we explorer a different checksum scheme that allows the learners to discard any invalid messages with minimal effort, this with only a low communication overhead for the replicas and learners involved. Consequently, the learners can directly select the appropriate messages and perform only a single information dispersal decode step. Inspired by the *fingerprints* of Alon et al. [2, 3], we base our checksum scheme on *Merkle trees* [51].

▶ **Definition 3.6.** *Consider a block of* $\mathbf{n}$ *update decisions* $\mathbb{B} = \mathbb{J}_{\mathfrak{R}}[(j-1)\mathbf{n} : \mathbf{n}]$*. The replica* $\textsc{r}$ *with* $\mathsf{id}(\textsc{r}) = i$*,* $0 \leq i < \mathbf{n}$*, should produce the* $i$*-th encoded piece* $\mathbb{B}_i = \mathtt{slice}_{\textsc{r}}(\mathbb{B})$*. To simplify presentation, we assume that the total number of such pieces is a power-of-two (otherwise, we simply add* `null`*-pieces until we have a power-of-two number of pieces). A* Merkle tree *build over these pieces is a balanced binary tree constructed as follows:*

1. *The* $i$*-th leaf of the tree has the value* $\mathtt{hash}(\mathbb{B}_i)$*.*
2. *The value of an internal node of which the left-child has value* $w_1$ *and the right-child has value* $w_2$ *is* $\mathtt{hash}([w_1, w_2])$*.*

[1] A theoretical attack of this type can always be detected by the learner: this attack will yield at least two sets of $\mathbf{g}$ pieces that decode to values $w$ and $\mathbb{B}$ with $w \neq \mathbb{B}$ and $\mathtt{hash}(w) = \mathtt{hash}(\mathbb{B})$.

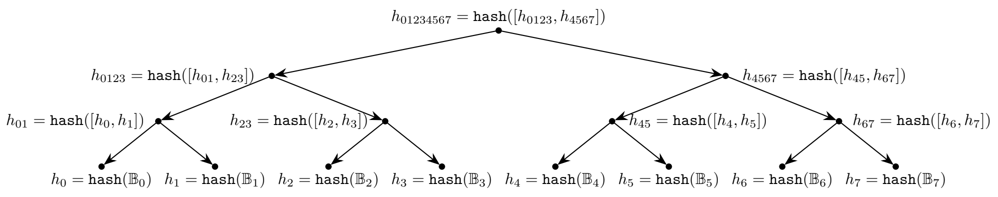
$h_{01234567} = \texttt{hash}([h_{0123}, h_{4567}])$
$h_{0123} = \texttt{hash}([h_{01}, h_{23}])$
$h_{4567} = \texttt{hash}([h_{45}, h_{67}])$
$h_{01} = \texttt{hash}([h_0, h_1])$
$h_{23} = \texttt{hash}([h_2, h_3])$
$h_{45} = \texttt{hash}([h_4, h_5])$
$h_{67} = \texttt{hash}([h_6, h_7])$
$h_0 = \texttt{hash}(\mathbb{B}_0)$
$h_1 = \texttt{hash}(\mathbb{B}_1)$
$h_2 = \texttt{hash}(\mathbb{B}_2)$
$h_3 = \texttt{hash}(\mathbb{B}_3)$
$h_4 = \texttt{hash}(\mathbb{B}_4)$
$h_5 = \texttt{hash}(\mathbb{B}_5)$
$h_6 = \texttt{hash}(\mathbb{B}_6)$
$h_7 = \texttt{hash}(\mathbb{B}_7)$

**Figure 5** A *Merkle tree* over eight data pieces $\mathbb{B}_0, \dots, \mathbb{B}_7$. The leaf nodes are each labeled with the hash of a data piece, while every internal node is labeled with the hash of the value of its two children. The *tree checksum* for $\mathbb{B}_5$ is $\texttt{checksum}(\mathbb{B}_5) = [h_{01234567}, h_{0123}, h_{67}, h_4]$.

*Notice that this construction is deterministic. Hence, every non-faulty replica will construct exactly the same Merkle tree for* $\mathbb{B}$. *The* tree checksum *we propose for the i-th piece* $\mathbb{B}_i$, $\texttt{checksum}(\mathbb{B}_i)$, *consists of the value of the root of the Merkle tree and the values of the sibling of each node on the path from the root to the i-th leaf.*

We illustrate this further in the following example.

▶ **Example 3.7.** Assume $\mathbf{n} = 8$ and consider a block $\mathbb{B}$ that encodes into pieces $\mathbb{B}_0, \dots, \mathbb{B}_7$. The Merkle tree for $\mathbb{B}$ can be found in Figure 5. The tree checksum $\texttt{checksum}(\mathbb{B}_5)$ is obtained as follows. First, the path from the root of the tree to the 5-th leaf visits the nodes with values $h_{4567}$, $h_{45}$, and $h_5$. The node with value $h_{4567}$ has the sibling with value $h_{0123}$; the node with value $h_{45}$ has the sibling with value $h_{67}$; and, finally, the node with value $h_5$ has the sibling with value $h_4$. The root of the tree has value $h_{01234567}$. Hence, $\texttt{checksum}(\mathbb{B}_5) = [h_{01234567}, h_{0123}, h_{67}, h_4]$.

Next, we show that these tree checksums are sufficient to recognize messages corrupted by Byzantine replicas.

▶ **Theorem 3.8.** *Consider the learner* $\textsc{l} \in \mathfrak{L}$ *after it has received all messages sent by the information dispersal steps following the* $\rho$*-th decision in* $\mathbb{J}_{\mathfrak{R}}$, $\rho \geq 1$. *If* $\mathbf{g} > \mathbf{b}$, *then, after receiving these messages,* $\textsc{l}$ *can reconstruct the first* $\mathbf{n}\lfloor \rho/\mathbf{n} \rfloor$ *update decisions made by* $\mathfrak{R}$ *using only* $\lfloor \rho/\mathbf{n} \rfloor$ *information dispersal decode steps.*

**Proof.** Let $i = \mathbf{n}\lfloor \rho/\mathbf{n} \rfloor$ be the last round after which $\textsc{l}$ received update decisions. As in the proof of Theorem 3.4, we only focus on how $\textsc{l}$ can reconstruct the block $\mathbb{B}$ containing update decisions $(i-1)\mathbf{n}, \dots, i\mathbf{n} - 1$, this independent of the behavior of the Byzantine replicas. Every message sent by non-faulty replicas will include a valid tree checksums. Each of these checksums is constructed over the same Merkle tree. Consequently, each of these checksums share the same value for the root of the Merkle tree. Hence, using the reasoning of Theorem 3.4, $\textsc{l}$ can reliably learn the root value $r$ of the Merkle tree after receiving at least $\mathbf{b} + 1$ messages with identical root values in their checksum.

Now consider the message $(i, s_j, c_j)$ received from the replica $\textsc{r}$ with $\mathsf{id}(\textsc{r}) = j$. To determine whether this message is valid and uncorrupted, we first check whether the root value in $c_j$ matches $r$. If this check fails, we can already discard the message. Next, we compute the hash $\texttt{hash}(s_j)$ to obtain the value of the $j$-th leaf in the Merkle tree. We observe that $c_j$ contains the value of the sibling of the $j$-th leaf. Hence, we can construct the value of the parent $p$ of the $j$-th leaf. This can be repeated: for any ancestor of the $j$-th leaf, $c_j$ also contains the value of the sibling of this ancestor. Hence, one can recompute the value of every ancestor of the $j$-th leaf based on the value $s_j$. When done, one will obtain the root value $r$ when the message is valid and uncorrupted. If any other value is obtained, then the message must be corrupted and one can discard the message.

As with the simple checksums, the only way in which Byzantine replicas can subvert the learning step is by finding hash collisions. Hence, assuming reasonable limits on the computational resources, the Byzantine replicas are unable to subvert the learning step. ◀

To further clarify the verification of messages, we illustrate how the verification process of the proof of Theorem 3.8 works:

▶ **Example 3.9.** Consider the situation of Example 3.7. Let L be a learner that already determined that the root value is $h_{01234567}$. At some point, L receives a message containing $\mathbb{B}'_5$ and the checksum $\texttt{checksum}(\mathbb{B}'_5) = [w_1, w_2, w_3, w_4]$ from replica R with $\mathsf{id}(\text{R}) = 5$. The learner checks whether $w_1 = h_{01234567}$, as otherwise the message is discarded. We assume $w_1 = h_{01234567}$. Next, L computes

$$\begin{aligned} h'_5 &= \texttt{hash}(\mathbb{B}'_5);\\ h'_{45} &= \texttt{hash}([w_4, h'_5]);\\ h'_{4567} &= \texttt{hash}([h'_{45}, w_3]);\\ h'_{01234567} &= \texttt{hash}([w_2, h_{4567}]). \end{aligned}$$

If $\mathbb{B}'_5 = \mathbb{B}_5$, $w_4 = h_4$, $w_3 = h_{67}$, and $w_2 = h_{0123}$, then $h'_{01234567} = h_{01234567}$ and the message received from R is valid and uncorrupted. In any other case, the resulting value $h'_{01234567}$ will not match $h_{01234567}$ and the message is discarded.

In Theorem 3.8, we analyzed the computational complexity of the information learning step with tree checksums in terms of the number of information dispersal decode steps. As we show in Example 3.9, one also needs to validate the correctness of each message via its tree checksum, for which $\log(\mathbf{n})$ hashes need to be computed. In practice, the information dispersal decode steps are much more costly than these validation steps (this is especially true when using modern processors that provide hardware acceleration for hashing). Hence, in our analysis, we only focus on the number of information dispersal decode steps.

Notice that, for any block $\mathbb{B}$, we obtain $\|\texttt{checksum}(\mathbb{B})\| = \Theta(\log(\mathbf{n}))$, this independent of $\|\mathbb{B}\|$. By combining Theorem 3.2 and Theorem 3.8, we obtain:

▶ **Corollary 3.10.** *Consider the learner* L $\in \mathfrak{L}$ *and replica* R $\in \mathcal{G}$*. If* $\mathbf{g} > \mathbf{b}$*, then the delayed-replication algorithm with tree checksums guarantees*

1. L *will learn the update journal* $\mathbb{J}_{\mathfrak{R}}$*;*
2. L *will receive at most* $|\mathbb{J}_{\mathfrak{R}}|$ *messages with a total size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|(\mathbf{n}/\mathbf{g}) + |\mathbb{J}_{\mathfrak{R}}|\log(\mathbf{n}))$*;*
3. L *will only need at most* $|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n}$ *information dispersal decode steps; and*
4. R *will sent at most* $|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n}$ *messages to* L *with a total size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g} + (|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n})\log(\mathbf{n}))$.

## 4 Use Case: Improving Permissioned Blockchains

In this paper, we introduced the *Byzantine learner problem* and the *delayed-replication algorithm*, this to support the *hierarchical architecture* for fault-tolerant and federated data management systems that we envisioned in Section 1.1. Our hierarchical architecture relies on a Byzantine cluster to manage the data. Typically, such Byzantine clusters are implemented by permissioned fully-replicated blockchains that use traditional consensus techniques. Next, we illustrate how the delayed-replication algorithm can be generalized to improve on such permissioned blockchains by introducing *scalable shared storage* instead of full replication and by reducing the cost of *update decision making*. Consequently, our techniques also improves the applicability of permissioned blockchains to extend database systems towards fault-tolerant and federated data management.

## 4.1 Towards Scalable Shared Storage

As noted in the Introduction, state-of-the-art systems use fully replicated designs in which every replica in a cluster $\mathfrak{R}$ maintains the full update journal $\mathbb{J}_{\mathfrak{R}}$. Replicas $\text{R} \in \mathfrak{R}$ typically only need the current view $\mathbb{V}$ of the data to make update decisions, however, and do not need access to the full history of all updates stored in $\mathbb{J}_{\mathfrak{R}}$. Hence, a fully replicated design is unnecessary costly and limits scalability. Fortunately, the delayed-replication algorithm already showed that full replication of $\mathbb{J}_{\mathfrak{R}}$ is unnecessary to guarantee the ability to recover and learn $\mathbb{J}_{\mathfrak{R}}$. Instead of storing all of $\mathbb{J}_{\mathfrak{R}}$ at each R, each replica R can simply processes each block $\mathbb{B}$ of $\mathbf{n}$ journal updates, compute $\texttt{slice}_{\text{R}}(\mathbb{B})$, and only keep this encoded piece around. This lowers the storage cost for $\mathbb{J}_{\mathfrak{R}}$ from $\|\mathbb{J}_{\mathfrak{R}}\|$ per replica to $\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g}$ per replica, which makes the storage capacity of $\mathfrak{R}$ scalable with the number of non-faulty replicas without hampering the availability of $\mathbb{J}_{\mathfrak{R}}$ for replica recovery and for external learners.

▶ **Example 4.1.** Consider a federated inventory management system $(\mathfrak{R}, \mathfrak{L})$ used by several companies to keep track of their inventories and of transactions between them. To decide upon the updates on the data, replicas in $\mathfrak{R}$ only need to be able to validate updates: e.g., a transfer of ownership from company $A$ to company $B$ of a product is only a valid update if $A$ originally owned the product. Hence, for validation, it is not necessary that replicas in $\mathfrak{R}$ maintain full copies of the journal $\mathbb{J}_{\mathfrak{R}}$, they only need the status of the current inventory, a much smaller dataset. Other tasks such as periodic analytics and data provenance will need read-only access to the full history of the data, which they can obtain as learners via the delayed-replication algorithm.

To further illustrate the necessity of storage scalability in blockchains, we only have to look at the permissionless Bitcoin blockchain. The size of the Bitcoin *ledger*, which represents a fully-replicated journal of financial transactions, is currently exceeding 256 GB and has grown with 59 GB over the last year. As noted in the introduction, Bitcoin is only able to process 7 transactions per second whereas Visa already processes 2000 transactions per second on average [61]. The permissioned blockchains our work focusses on can easily process hundreds to thousands transactions per second, as already exemplified by the BFS system in 2002 [14, 15]. Hence, the size of the journal maintained by permissioned blockchains can grow even more rapidly. We conclude:

▶ **Proposition 4.2.** *Let $\mathfrak{R}$ be a Byzantine cluster with update journal $\mathbb{J}_{\mathfrak{R}}$, current data view $\mathbb{V}$, and in which only $\mathbb{V}$ is necessary to make update decisions. If $\mathbf{g} > \mathbf{b}$, then the delayed-replication algorithm can provide storage scalability with these guarantees:*

1. *If simple checksums are used, then the storage cost per replica $\text{R} \in \mathfrak{R}$ is reduced from $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\| + \|\mathbb{V}\|)$ to $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g} + \|\mathbb{V}\|)$.*
2. *If tree checksums are used, then the storage cost per replica $\text{R} \in \mathfrak{R}$ is reduced from $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\| + \|\mathbb{V}\|)$ to $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g} + (|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n})\log(\mathbf{n}) + \|\mathbb{V}\|)$.*

**Proof.** These results follow directly from Corollaries 3.5 and 3.10. ◀

## 4.2 Improved Checkpoints in Byzantine Consensus

Next, we will show how the delayed-replication algorithm can be used internally in Byzantine clusters to reduce the cost of decision making. Typical permissioned blockchains use *consensus protocols* to coordinate making update decisions [4, 8, 9, 10, 14, 15, 17, 28, 34, 42, 44, 45, 48, 69, 71]. Most practical consensus protocols can be traced back to the influential design of the *Practical Byzantine Fault Tolerance protocol* (`Pbft`) of Castro et al. [14, 15].

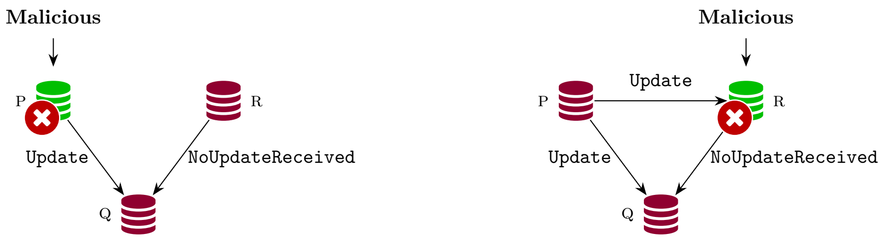

**Figure 6** Two cases of possible malicious behavior. On the left, the primary P is malicious. On the right, the replica R is malicious. Unfortunately, the non-faulty replica Q receives the same set of messages in both cases and, hence, cannot determine which replica is malicious in which case.

In `Pbft`, a replica is elected *primary* and is in charge of coordinating update decision making among all replicas. To allow other replicas to determine whether the primary, which could be Byzantine, is coordinating correctly, `Pbft` employs two phases of broadcast-based communication among all replicas before any update decision is committed. This communication ensures that update decisions are only committed if a majority of all non-faulty replicas are aware of these decisions. Unfortunately, a malicious primary can keep some non-faulty replicas *in the dark* by not sending them any update decisions. The remaining Byzantine replicas can cover for this malicious behavior by acting as non-faulty replicas. Such an attack cannot be reliably detected by the other replicas in the system, as the following example illustrates.

▶ **Example 4.3.** Let $\mathfrak{R}$ be a Byzantine cluster with $\mathbf{n} = 3\mathbf{b} + 1$. Let $\mathrm{P} \in \mathfrak{R}$ be the elected primary, let $\mathrm{R} \in \mathfrak{R}$ be another replica, and let $\mathrm{Q} \in \mathcal{G}$ be a non-faulty replica. Assume that a correct primary will send the same updates to all replicas. Consider the following two cases:

1. We have $\mathrm{R} \in \mathcal{G}$ and $\mathrm{P} \in \mathcal{B}$. The primary sends updates via `Update` messages, except that it excludes R. The replica R detects this, as it does not receive any messages, and notifies Q that the primary is malicious via a `NoUpdateReceived` message.
2. We have $\mathrm{R} \in \mathcal{B}$ and $\mathrm{P} \in \mathcal{G}$. Independent of the actions of the primary, R notifies Q that the primary is malicious via a `NoUpdateReceived` message.

We have sketched these two cases in Figure 6. In both cases, Q receives exactly the same set of messages from P and R. Consequently, Q cannot determine which of the replicas is malicious and which of the replicas is non-faulty.

The issue of replicas being left in the dark is faced by not only `Pbft`, but also by many other practical primary-backup protocols. The typical way to assure that all replicas eventually learn the update decisions made, even when the primary is malicious, is by using periodical checkpoints. Unfortunately, the usual checkpoint protocols employed use broadcast-based primitives with very high communication complexity. Moreover, checkpoint protocols typically have a pull-based component, which makes them vulnerable to the attacks illustrated in Example 2.5. Fortunately, we can employ the push-based delayed-replication algorithm to provide checkpoints with low costs for all replicas involved. To do so, we model any replica left in the dark by a malicious primary as *crashed*. In the worst case, `Pbft` allows for a situation in which $\mathbf{g} = \mathbf{b}+1$ and $\mathbf{c} = \mathbf{b}$ (the maximum number of non-Byzantine replicas a malicious primary can keep in the dark without being detected), making $\mathbf{n} \geq 3\mathbf{b}+1$. Hence, in all situations our delayed-replication algorithm can be employed as a checkpoint protocol by making all replicas in $\mathfrak{R}$ listeners. We conclude:

▶ **Proposition 4.4.** *Let $\mathfrak{R}$ be a Byzantine cluster running* `Pbft`. *If $\mathbf{g} > \mathbf{b}$, then the delayed-replication algorithm can provide* `Pbft`*-style checkpoints with these guarantees:*

1. *If simple checksums are used, then every replica* $\textsc{r} \in \mathfrak{R}$ *will be able to learn* $\mathbb{J}_{\mathfrak{R}}$ *and will send and receive at most* $|\mathbb{J}_{\mathfrak{R}}|$ *messages with a combined size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|)$.
2. *If tree checksums are used, then every replica* $\textsc{r} \in \mathfrak{R}$ *will be able to learn* $\mathbb{J}_{\mathfrak{R}}$ *and will send and receive at most* $|\mathbb{J}_{\mathfrak{R}}|$ *messages with a combined size of* $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\| + |\mathbb{J}_{\mathfrak{R}}| \log(\mathbf{n}))$.

**Proof.** We choose $\mathfrak{L} = \mathfrak{R}$ and we use the delayed-replication algorithm with $\mathbf{c} = \mathbf{b}$ and, as $\mathfrak{R}$ is running `Pbft`, $\mathbf{n} > 3\mathbf{b}$. Hence, $\mathbf{g} = \mathbf{n} - \mathbf{c} - \mathbf{b} = \mathbf{n} - 2\mathbf{b} \geq \mathbf{b} + 1$ non-faulty replicas will be senders in the delayed-replication algorithm, guaranteeing success.

Next, we consider the number of messages sent and received. First, consider the case using simple checksums. Let $\textsc{r} \in \mathcal{G}$ be a non-faulty replica that is not left in the dark. Applying the results of Corollaries 3.5, we learn that replica $\textsc{r}$ sends at most $|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n}$ messages to each replica with a total size of $\mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g})$. As $\mathbf{n} = |\mathfrak{R}|$, $\textsc{r}$ will send $\mathbf{n}(|\mathbb{J}_{\mathfrak{R}}|/\mathbf{n}) = |\mathbb{J}_{\mathfrak{R}}|$ messages with a total size of at most $d\mathbf{n}(\|\mathbb{J}_{\mathfrak{R}}\|/\mathbf{g})$, for some constant $d$. We have $\mathbf{n} > 3\mathbf{b}$ and $\mathbf{g} = \mathbf{n} - 2\mathbf{b}$. Hence, $\mathbf{n} = 3\mathbf{b} + j$ and $\mathbf{g} = \mathbf{n} - 2\mathbf{b} = \mathbf{b} + j$ for some $j$, $j \geq 1$. We have $\mathbf{n}/\mathbf{g} = (3\mathbf{b} + j)/(\mathbf{b} + j) = 1 + 2\mathbf{b}/(\mathbf{b} + j)$. As $j \geq 1$, we have $0 \leq 2\mathbf{b}/(\mathbf{b} + j) < 2$ and $\mathbf{n}/\mathbf{g} \leq 3$. We conclude that the total size of all messages send by $\textsc{r}$ is upper bounded by $3d\|\mathbb{J}_{\mathfrak{R}}\| = \mathcal{O}(\|\mathbb{J}_{\mathfrak{R}}\|)$. In a similar manner, we can derive the same upper bounds on the number and size of the messages received by $\textsc{r}$. For the case using tree checksums, we applying the results of Corollary 3.10 to the above reasoning, which leads to only adding a cost of $\mathcal{O}(\log(\mathbf{n}))$ to each message sent and received. ◀

Since the introduction of `Pbft`, many improvements on its design have been proposed. Recently, there has been a surge in protocols that aim at bringing down the communication cost of the normal-case operations of `Pbft` from a *quadratic amount* per update decision to a *linear amount*, which vastly improves the scalability of consensus. Examples include `HotStuff` [75], `LinBFT` [74], and `SBFT` [33]. These examples all use *threshold signatures* [66] to summarize confirmation of any decision by the majority of all replicas in a constant-sized signature—which eliminates the need for broadcast-based quadratic communication among all replicas. None of the current approaches satisfactory deal with recovery of replicas that are left in the dark, however. Hence, we believe that our highly-efficient delayed-replication algorithm can fill in the checkpoint gap in such linear designs.

## 5 Related Work

There is an abundant literature on distributed systems, distributed scalable storage (e.g., via information dispersal), and on fault-tolerant fully-replicated systems (e.g. [12, 60, 69, 70]). In this paper, we primarily focused on bridging the gap between, on the one hand, fault-tolerant systems and, on the other hand, scalable distributed systems.

**Learners in fault-tolerant systems.** `Paxos`, a consensus protocol that can be used to make reliable update decisions in a cluster with only crash failures, and several `Paxos`-like protocols have a concept of *learners* [46, 47, 49]. As the name suggests, these `Paxos`-learners will learn all update decisions made by the cluster, not unlike the learners we propose. In `Paxos`, these learners are also crucial to determine whether consensus is reached, are deeply involved in the consensus protocol, cannot be arbitrarily selected, and perform significant amounts of communication, however. This makes the architecture of `Paxos` incomparable with the hierarchical architecture we propose.

Most other consensus protocols, especially those based on the *Practical Byzantine Fault Tolerance protocol* (`Pbft`) of Castro et al. [14, 15], do not distinguish between the roles of replicas in a Byzantine cluster. In `Pbft`, some *read-only* optimizations are considered, but even these optimizations require participation of all replicas involved. Hence, the approach of `Pbft` towards read-only data processing is non-scalable, whereas our hierarchical architecture benefits from the addition of non-faulty replicas to the Byzantine cluster.

The HyperLedger permission blockchain fabric does utilize a hierarchical design similar to what we propose [6]. HyperLedger distinguishes between, on the one hand, endorsers and orderers that coordinate data updates, and, on the other hand, peers that only learn of data updates. Currently, HyperLedger relies on Apache Kafka [26] to provide only crash-tolerant ordering and to broadcast updates to peers. The approach we propose—by using the delayed-replication algorithm—is not only able to tolerate an arbitrary number of crashes, but also addresses Byzantine behavior. Furthermore, our approach is highly scalable and requires only a minimum of communication. Finally, our approach enables a way towards permissioned blockchains with scalable storage, which is not provided by HyperLedger.

Finally, the Byzantine learner problem we study in this paper differs from the *cluster sending problem* of Hellings et al. [37]. On the one hand, we provide in this work techniques to stream sequences of data updates from a single Byzantine cluster to learners, this with minimal communication costs in terms of the data exchanged. On the other hand, the cluster sending problem of Hellings et al. [37] focusses on the problem of sending a single value between two Byzantine clusters with minimal communication costs in terms of the number of messages exchanged.

**Information dispersal and scalable storage.** `IDA`, the information dispersal algorithm we utilize in the delayed-replication algorithm, was proposed by Rabin [63] to provide reliable load-balanced storage and communication in a distributed setting. Alon et al. [2, 3] expanded `IDA` towards recovery of failures by adapting the scheme used in `IDA` towards recognizing faulty encoded pieces. Unfortunately, the methods employed by Alon et al. always introduce a space overhead per participant. We build upon these information dispersal techniques by using them to solve the *Byzantine learner problem* and we showed how these techniques can be used to resolve current issues in state-of-the-art permissioned blockchains that provide fault-tolerant and federated data management. Moreover, our results structurally improve on the results of Rabin and Alon et al. via the delayed-replication algorithm with simple checksums, which enables Byzantine fault-tolerant communication and storage without any space overhead.

## 6 Conclusions and Future Work

In this paper, we studied the *Byzantine learner problem*—the problem of efficiently distributing a sequence of data updates made by replicas in a Byzantine cluster to an arbitrary number of learners. As our main result, we proposed the *delayed-replication algorithm* to address this Byzantine learner problem. Our algorithm is coordination-free, equally distributes communication costs among all replicas, and leverages information dispersal to achieve Byzantine learning with minimal communication costs. Our delayed-replication algorithm opens the door to hierarchical fault-tolerant and federated database systems that can effectively deal with big read-only workloads, e.g., by running complex data processing tasks on individual specialized learners and by providing trusted read-only proxies close to end-users for fast query answering. Moreover, we showed that the delayed-replication algorithm and its underlying techniques open the door for the development of new high-performance fault-tolerant database systems by improving the design of existing permissioned blockchain-based database systems.

Our techniques are only a first step toward developing fault-tolerant, reliable, scalable, and high-performance database systems and permissioned blockchains. To further these developments, we need not only support big read-only workloads and storage scalability, but also improve on other areas. To do so, we see several avenues of future research and development:

1. Our techniques can be used to improve fault-tolerant and federated data management by reducing the cost of read-only workloads and by introducing scalable shared storage. We did not yet address the scalability of decision making (e.g., write workloads), however. To improve decision making, we are interested in the development of efficient decision making techniques in Byzantine settings that—for performance reasons—provides less strict guarantees than traditional consensus-based techniques. To further aid scalability of decision making, we are also interested in developing further non-fully-replicated designs, e.g., by incorporating fault-tolerant sharding.
2. Our design is primarily intended to reduce the cost of read-only workloads that require access to the full history of changes. Examples of such workloads include analytics, data provenance, machine learning, visualization, and read-only proxies. Besides these workloads, there are also many smaller read-only workloads, e.g., one-off relational querying. Current fault-tolerant approaches toward such workloads remain non-scalable, as they require the independent execution of such queries by all replicas in the Byzantine cluster. We are interested in the development of techniques that can lift this burden on scalability.

## References

1 2ndQuadrant. Postgres-XL: Open source scalable SQL database cluster. URL: `https://www.postgres-xl.org/`.

2 Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien Stern. Scalable secure storage when half the system is faulty. *Information and Computation*, 174(2):203–213, 2002. `doi:10.1006/inco.2002.3148`.

3 Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien Stern. Addendum to "scalable secure storage when half the system is faulty". *Information and Computation*, 205(7):1114–1116, 2007. `doi:10.1016/j.ic.2006.02.007`.

4 Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010. `doi:10.1109/TDSC.2008.53`.

5 Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019. `doi:10.14778/3342263.3342275`.

6 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018. `doi:10.1145/3190508.3190538`.

7 GSM Association. Blockchain for development: Emerging opportunities for mobile, identity and aid, 2017. URL: `https://www.gsma.com/mobilefordevelopment/wp-content/uploads/2017/12/Blockchain-for-Development.pdf`.

8 Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4):12:1–12:45, 2015. `doi:10.1145/2658994`.

9 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013. `doi:10.1109/ICDCS.2013.53`.

10 Christian Berger and Hans P. Reiser. Scaling byzantine consensus: A broad analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL'18, pages 13–18. ACM, 2018. `doi:10.1145/3284764.3284767`.

11 Burkhard Blechschmidt. Blockchain in Europe: Closing the strategy gap. Technical report, Cognizant Consulting, 2018. URL: `https://www.cognizant.com/whitepapers/blockchain-in-europe-closing-the-strategy-gap-codex3320.pdf`.

12 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *31st International Symposium on Distributed Computing*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.DISC.2017.1`.

13 Michael Casey, Jonah Crane, Gary Gensler, Simon Johnson, and Neha Narula. The impact of blockchain technology on finance: A catalyst for change. Technical report, International Center for Monetary and Banking Studies, 2018. URL: `https://www.cimb.ch/uploads/1/1/5/4/115414161/geneva21_1.pdf`.

14 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

15 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002. `doi:10.1145/571637.571640`.

16 Christie's. Major collection of the fall auction season to be recorded with blockchain technology, 2018. URL: `https://www.christies.com/presscenter/pdf/9160/RELEASE_ChristiesxArtoryxEbsworth_9160_1.pdf`.

17 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168. USENIX Association, 2009.

18 Cindy Compert, Maurizio Luinetti, and Bertrand Portier. Blockchain and GDPR: How blockchain could address five areas associated with gdpr compliance. Technical report, IBM Security, 2018. URL: `https://public.dhe.ibm.com/common/ssi/ecm/61/en/61014461usen/security-ibm-security-solutions-wg-white-paper-external-61014461usen-20180319.pdf`.

19 Oracle Corporation. Maximize availability with Oracle Database 18c. URL: `https://www.oracle.com/technetwork/database/availability/maximum-availability-wp-18c-4403435.pdf`.

20 Oracle Corporation. MySQL - MySQL 8.0 reference manual: 17 replication. URL: `https://dev.mysql.com/doc/refman/8.0/en/replication.html`.

21 Alex de Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018. `doi:10.1016/j.joule.2018.04.016`.

22 Microsoft Docs. SQL Server replication. URL: `https://docs.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication`.

23 Wayne W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. Technical report, The Data Warehousing Institute, 101communications LLC., 2002.

24 Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019. `doi:10.14778/3342263.3342636`.

25 Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Arvind Arasu, and Donald Kossmann. BlockchainDB – towards a shared database on blockchains. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1905–1908. ACM, 2019. doi:10.1145/3299869.3320237.

26 The Apache Software Foundation. Apache Kafka: A distributed streaming platform. URL: https://kafka.apache.org/.

27 Lan Ge, Christopher Brewster, Jacco Spek, Anton Smeenk, and Jan Top. Blockchain for agriculture and food: Findings from the pilot study. Technical report, Wageningen University, 2017. URL: https://www.wur.nl/nl/Publicatie-details.htm?publicationId=publication-way-353330323634.

28 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.

29 William J. Gordon and Christian Catalini. Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. *Computational and Structural Biotechnology Journal*, 16:224–230, 2018. doi:10.1016/j.csbj.2018.06.003.

30 Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978. doi:10.1007/3-540-08755-9_9.

31 Andy Greenberg. The untold story of NotPetya, the most devastating cyberattack in history, 2018. URL: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

32 The PostgreSQL Global Development Group. PostgreSQL documentation: Chapter 26. high availability, load balancing, and replication. URL: https://www.postgresql.org/docs/current/static/high-availability.html.

33 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure, 2019. URL: https://arxiv.org/abs/1804.01626.

34 Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.44.

35 Suyash Gupta and Mohammad Sadoghi. *Blockchain Transaction Processing*, pages 1–11. Springer International Publishing, 2018. doi:10.1007/978-3-319-63962-8_333-1.

36 Suyash Gupta and Mohammad Sadoghi. EasyCommit: A non-blocking two-phase commit protocol. In *Proceedings of the 21st International Conference on Extending Database Technology*. Open Proceedings, 2018. doi:10.5441/002/edbt.2018.15.

37 Jelle Hellings and Mohammad Sadoghi. Brief announcement: The fault-tolerant cluster-sending problem. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.45.

38 Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.

39 Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data Quality and Record Linkage Techniques*. Springer New York, 2007. doi:10.1007/0-387-69505-2.

40 Matt Higginson, Johannes-Tobias Lorenz, Björn Münstermann, and Peter Braad Olesen. The promise of blockchain. Technical report, McKinsey&Company, 2017. URL: https://www.mckinsey.com/industries/financial-services/our-insights/the-promise-of-blockchain.

41 Maged N. Kamel Boulos, James T. Wilson, and Kevin A. Clauson. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *International Journal of Health Geographics*, 17(1):1211–1220, 2018. doi:10.1186/s12942-018-0144-x.

42 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 295–308. ACM, 2012. doi:10.1145/2168836.2168866.

43 Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2nd edition, 2014.

44 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*, pages 45–58. ACM, 2007. doi:10.1145/1294261.1294267.

45 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, 2009. doi:10.1145/1658357.1658358.

46 Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978. doi:10.1016/0376-5075(78)90045-4.

47 Leslie Lamport. Paxos made simple. *ACM SIGACT News, Distributed Computing Column 5*, 32(4):51–58, 2001. doi:10.1145/568425.568433.

48 Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2019. doi:10.1109/TC.2018.2860009.

49 Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006. doi:10.1109/TDSC.2006.35.

50 Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1st edition, 1996.

51 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer Berlin Heidelberg, 1988. doi:10.1007/3-540-48184-2_32.

52 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: https://bitcoin.org/en/bitcoin-paper.

53 Arvind Narayanan and Jeremy Clark. Bitcoin's academic pedigree. *Communications of the ACM*, 60(12):36–45, 2017. doi:10.1145/3132259.

54 Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, 2019. doi:10.14778/3342263.3342632.

55 Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019. doi:10.1109/ICDE.2019.00020.

56 Dick O'Brie. Internet security threat report: Ransomware 2017, an ISTR special report. Technical report, Symantec, 2017. URL: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-ransomware-2017-en.pdf.

57 The Council of Economic Advisers. The cost of malicious cyber activity to the U.S. economy. Technical report, Executive Office of the President of the United States, 2018. URL: https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf.

58 National Audit Office. Investigation: Wannacry cyber attack and the NHS, 2018. URL: https://www.nao.org.uk/report/investigation-wannacry-cyber-attack-and-the-nhs/.

59 Michael Okun. *Byzantine Agreement*, pages 255–259. Springer New York, 2016. doi:10.1007/978-1-4939-2864-4_60.

60 M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer New York, 3th edition, 2011.

61 Michael Pisa and Matt Juden. Blockchain and economic development: Hype vs. reality. Technical report, Center for Global Development, 2017. URL: `https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality`.

62 PwC. Blockchain – an opportunity for energy producers and consumers?, 2016. URL: `https://www.pwc.com/gx/en/industries/energy-utilities-resources/publications/opportunity-for-energy-producers.html`.

63 Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989. `doi:10.1145/62044.62050`.

64 Thomas C. Redman. The impact of poor data quality on the typical enterprise. *Communications of the ACM*, 41(2):79–82, 1998. `doi:10.1145/269012.269025`.

65 Scott Ruoti, Ben Kaiser, Arkady Yerukhimovich, Jeremy Clark, and Robert Cunningham. Blockchain technology: What is it good for? *Communications of the ACM*, 63(1):46—-53, 2019. `doi:10.1145/3369752`.

66 Victor Shoup. Practical threshold signatures. In *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220. Springer Berlin Heidelberg, 2000. `doi:10.1007/3-540-45539-6_15`.

67 Dale Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

68 Symantec. Internet security threat report, volume 32, 2018. URL: `https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf`.

69 Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

70 Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3th edition, 2017. URL: `https://www.distributed-systems.net/`.

71 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013. `doi:10.1109/TC.2011.221`.

72 Harald Vranken. Sustainability of bitcoin and blockchains. *Current Opinion in Environmental Sustainability*, 28:1–9, 2017. `doi:10.1016/j.cosust.2017.04.011`.

73 Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. EIP-150 revision. URL: `https://gavwood.com/paper.pdf`.

74 Yin Yang. LinBFT: Linear-communication byzantine fault tolerance for public blockchains, 2018. URL: `https://arxiv.org/abs/1807.01829`.

75 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019. `doi:10.1145/3293611.3331591`.