Federated Byzantine Quorum Systems

Álvaro García-Pérez
IMDEA Software Institute, Madrid, Spain
alvaro.garcia.perez@imdea.org
https://orcid.org/0000-0002-9558-6037

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain
alexey.gotsman@imdea.org

Abstract

Some of the recent blockchain proposals, such as Stellar and Ripple, aim to make trust assumptions flexible: they allow each node to select which other nodes it trusts. Unfortunately, the theoretical foundations underlying such blockchains have not been thoroughly investigated. To close this gap, in this paper we study the mechanism of specifying trust assumptions by means of federated Byzantine quorum systems (FBQS), used by Stellar. We rigorously prove the correctness of basic constructions over FBQS and demonstrate that they can be used to implement a Byzantine-fault-tolerant atomic register. We furthermore relate FBQS to the classical Byzantine quorum systems studied in distributed computing theory.

2012 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases blockchain, Stellar consensus protocol, byzantine fault tolerance, byzantine quorum systems, read/write register

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Blockchains are distributed databases that maintain a ledger over a set of potentially Byzantine nodes. The nodes use a Byzantine fault-tolerant consensus protocol to agree on a total order in which transactions are stored in the ledger. Blockchains usually come in two flavours. *Permissionless* blockchains allow anyone to participate, and are often based on consensus protocol such as proof-of-work and proof-of-stake. *Permissioned* blockchains assume a known set of participants, and are often based on classical BFT consensus protocols, such as PBFT [4]. However, some of the new permissioned blockchains, such as Stellar [13] and Ripple [14], have intriguing designs that use quorum-like structures typical for BFT consensus, yet allow the system to be open to participants. This is achieved by allowing each protocol participant to choose its trust assumptions separately. In particular, in Stellar these trust assumptions are specified using a *federated Byzantine quorum system*[1] (*FBQS* for short): each node participating in the blockchain can select a set of *quorum slices*: sets of nodes each of which would convince the node to accept a validity of a given statement. A set of nodes U such that each node in U has some quorum slice fully within U forms a *quorum*—a set of nodes that can potentially reach an agreement. The agreement on the blockchain is then maintained by a fairly intricate protocol, the core of which is *federated voting*, essentially solving a form of binary consensus.

[1] Called *federated Byzantine agreement systems* in the original [13]. The name used in this paper emphasises that their purpose is not restricted to solve consensus.

© Álvaro García-Pérez and Alexey Gotsman;
licensed under Creative Commons License CC-BY
42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Even though Stellar has been deployed as a functioning blockchain, its theoretical foundations remain shaky in several ways. First, the core protocols used in Stellar lack rigorous proofs of correctness or, for that matter, even useful statements of what correctness means. Second, even though Stellar is based on some general concepts for distributing trust, these concepts have only been applied in the context of a complete blockchain. This leaves it unclear whether the concepts are applicable more widely. Finally, as observed in [3], the quorums arising in Stellar are similar to well-studied *Byzantine quorum systems* [11], which can be used to solve problems beyond consensus, *e.g.*, *safe*, *regular* or *atomic register* [9]. However, so far the relationship between these has not been investigated.

In this paper, we aim to close these gaps and perform a rigorous theoretical study of concepts underlying the Stellar blockchain. To this end, we make the following contributions.

First, we rigorously state and prove the correctness of the federated voting protocol used by Stellar (Section 3.2). Stating correctness in a federated setting is nontrivial. Unlike in the classical Byzantine setting, where nodes can be correct of faulty, here correct nodes subdivide into two classes: *befouled* and *intact* [13]. Befouled nodes correctly follow the protocol, but choose their quorum slices in a way that allows faulty nodes to convince them of wrong statements; without due care in the protocol, this may lead befouled nodes to compute wrong results. Intact nodes are correct nodes that are not befouled. We prove that Stellar's federated voting protocol ensures that any pair of correct nodes, either befouled or intact, cannot report contradictory consensus results, except for the pathological situation where all nodes choose their slices in such a bad way that no node is intact (Theorem 5). This correctness statement is stronger than the one given in the Stellar proposal, which only provided such a guarantee for intact nodes. The difference is significant in practice: whereas, as a rule of thumb, one may assume a bound on the number of nodes that can be faulty at a time, such a bound cannot be easily given for befouled nodes. Hence, with a correctness statement restricting only the behaviour of intact nodes a client cannot easily ensure it gets correct results by querying a representative set of nodes. Unlike the existing correctness statement, ours additionally allows for faulty nodes to lie to others about their selection of quorum slices. We show that, even though this does affect the computation performed by other nodes, this may only hurt the node who lied and not others (Section 4).

Second, to demonstrate that the concept of FBQS is more generally applicable, we show how to implement a read/write register over an FBQS, whose safety is formalised by Byzantine fault-tolerant linearisability [10] and liveness by finite-write termination [1] (Section 5.3). Our protocol is inspired by federated voting and, as part of its proof, we show that executions it produces correspond to executions of federated voting.

Finally, we study the relationship between the FBQSs and the Byzantine quorum systems of [11]. We introduce a correspondence between an FBQS and the variant of a Byzantine quorum system called *dissemination quorum system* (DQS for short) in Section 5 of [11]. A DQS consists of a set of quorums, together with a system that characterises the failure scenarios that the DQS is tolerant to, called a *fail-prone system.* The correspondence between FBQSs and DQSs is one-to-many. An FBQS determines uniquely the set of quorums, and a collection of fail-prone systems that are compatible with the FBQS—*i.e.*, they characterise failure scenarios that the FBQS is also tolerant to. Off-the-shelf DQS algorithms can be run on an FBQS by fixing a fail-prone system from the ones compatible with the FBQS.

The full proofs of the theorems and lemmas in the paper are collected in the appendix.

2 System Model

The system consists of a set of client processes $\mathbf{C}$ and a set of server processes $\mathbf{V}$. We assume a Byzantine failure model—*i.e.*, faulty processes can deviate arbitrarily from their specification. We let $\mathbf{C} = \mathbf{C}_{ok} \cup \mathbf{C}_{bad}$ where $\mathbf{C}_{ok}$ is the set of correct clients and $\mathbf{C}_{bad}$ the set of faulty clients.

We assume an asynchronous distributed system where nodes are connected by a network that may delay messages and deliver them out of order. For simplicity, we assume the network eventually delivers all the messages, and it does not corrupt nor duplicate them.

Clients use unforgeable signatures to authenticate communication. We denote a datum d signed by client c as $\langle d \rangle_c$. We assume with probability one that no process in the system other than c can send $\langle d \rangle_c$, unless the process is repeating a signed datum that it received before (we assume clients do not leak private keys). These signatures can be verified with public keys that are known to every process.

Clients tag certain messages with random nonces that are unique. We assume with probability one that every two nonces that are ever picked—by the same or different clients—are different.

3 Federated Byzantine Quorum Systems

We consider *federated Byzantine quorum systems* (*FBQS* for short) from [13], which aim to make the trust assumptions of each node flexible. In Section 3.1 we rephrase the definitions and results from [13], and in Section 3.2 we introduce an implementation of federated voting that we will use as a reference for the implementation of the read/write register of Section 5. At the end of Section 3.2 we rigorously state our novel safety result (Theorem 5).

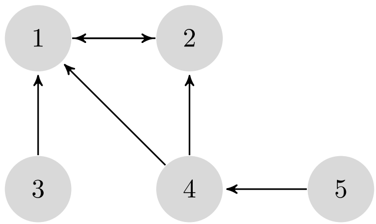
3.1 FBQSs Overview

An *FBQS* is a pair $\langle \mathbf{V}, \mathbf{Q} \rangle$ where $\mathbf{V}$ is a set of nodes and $\mathbf{Q} : \mathbf{V} \to 2^{2^{\mathbf{V}}} \setminus \{\emptyset\}$ is a quorum function specifying one or more *quorum slices* for each node, where a node belongs to all of its own quorum slices—*i.e.*, $\forall v \in \mathbf{V}.\ \forall q \in \mathbf{Q}(v).\ v \in q$.

Our FBQSs have the set of servers $\mathbf{V}$ as nodes, and from now on we will always refer to FBQS's nodes as 'servers'. Federated voting enforces flexible trust, since in an FBQS the servers have the freedom to trust any combination of parties that they see fit. The function $\mathbf{Q}$ for quorum slices reflects the choice of trust of each server. In the FBQSs of [13], servers do not lie about quorum slices and thus every server knows every other server's choice of trust. This situation is unrealistic because Byzantine servers may fail arbitrarily, and we address the issue of servers that lie about their quorum slices in Section 4.

A set of servers $U \subseteq \mathbf{V}$ in FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ is a *quorum* iff $U \neq \emptyset$ and U contains a slice for each member—*i.e.*, $\forall v \in U.\ \exists q \in \mathbf{Q}(v)$ such that $q \subseteq U$. A property that quorums must have in order to preserve safety is that of *quorum intersection*, which states that any two quorums share a server—*i.e.*, for all quorums U_1 and U_2, $U_1 \cap U_2 \neq \emptyset$. Another interesting property is that of *quorum availability*, which states that some quorum exists. Quorum availability is trivially met since the set $\mathbf{V}$ of servers is a quorum. An FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ that enjoys quorum intersection induces a *quorum system* $\mathcal{Q}$ *à la* Malkhi and Reiter [11] where $\mathbf{V}$ is the *universe* and $\mathcal{Q} = \{U \mid U \text{ is a quorum in } \langle \mathbf{V}, \mathbf{Q} \rangle\}$.

▶ **Example 1.** Consider the FBQS depicted below, where each server has only one slice, which is represented by the arrows departing from the server.



$$\begin{aligned}\mathbf{Q}(1) \;=\; \mathbf{Q}(2) \;&=\; \{\{1,2\}\}\\ \mathbf{Q}(3) \;&=\; \{\{1,3\}\}\\ \mathbf{Q}(4) \;&=\; \{\{1,2,4\}\}\\ \mathbf{Q}(5) \;&=\; \{\{4,5\}\}\end{aligned}$$

The FBQS meets quorum intersection—*i.e.*, all the quorums intersect at $\{1,2\}$—and quorum availability—*i.e.*, $\{1,2,3,4,5\}$ is a quorum—and thus it induces the quorum system

$$\mathcal{Q} = \{\{1,2\},\{1,2,3\},\{1,2,4\},\{1,2,3,4\},\{1,2,4,5\},\{1,2,3,4,5\}\}.$$

▶ **Example 2.** Consider the FBQS with $3f+1$ servers, where f is the threshold of fault tolerance, and were every server has a slice for each set of $2f+1$ servers. The FBQS induces a quorum system in which any set of $2f+1$ servers is a quorum.

Given a set $B \subseteq \mathbf{V}$ of servers, to *delete* B from $\langle \mathbf{V}, \mathbf{Q}\rangle$, written $\langle \mathbf{V}, \mathbf{Q}\rangle^B$, means to compute the modified FBQS $\langle \mathbf{V} \setminus B, \mathbf{Q}^B\rangle$ where $\mathbf{Q}^B(v) = \{q \setminus B \mid q \in \mathbf{Q}(v)\}$.

The notion of *dispensable set* defined below captures the tolerance of the system in the presence of a given set of faulty servers. Broadly, a dispensable set is a set of servers that can be deleted from the system while preserving quorum intersection and quorum availability.

Let $B \subseteq \mathbf{V}$ be a set of servers. We say B is a *dispensable set* (*DSet* for short) iff

(i) (*quorum intersection despite* B) $\langle \mathbf{V}, \mathbf{Q}\rangle^B$ enjoys quorum intersection, and

(ii) (*quorum availability despite* B) either $\mathbf{V} \setminus B$ is a quorum in $\langle \mathbf{V}, \mathbf{Q}\rangle$ or $B = \mathbf{V}$.

The inclusion of the trivial DSet $\mathbf{V}$ is justified in the cases where the failure of any server befouls the whole system, regardless of whether quorum intersection is preserved or not.

For instance, the set of DSets in the FBQS from Example 1 is

$$\mathcal{D} = \{\emptyset,\{3\},\{4,5\},\{5\},\{3,4,5\},\{1,2,3,4,5\}\}.$$

and the DSets in the FBQS from Example 2 consist of every set of f servers, together with the set $\mathbf{V}$ of all servers.

The DSets of an FBQS are determined *a priori* given each server's quorum slices, but which servers are correct or faulty depends on runtime behaviour. The DSets we care about are those that contain all faulty servers. The *befouled* servers are either faulty or they are correct but surrounded by too many befouled servers, which may convince them of wrong statements. The rest of the servers are *intact*. Formally, a server v is *intact* iff there exists a DSet B containing all faulty servers and such that $v \notin B$. Otherwise, v is *befouled*.

Assume that, in the FBQS from Example 1, server 4 is faulty and all the other servers are correct. Since 4 could, single-handedly, convince 5 to accept any statement, then 5 is correct but befouled. The DSets that contain all faulty servers are $\{4,5\}$, $\{3,4,5\}$ and $\{1,2,3,4,5\}$. The servers in $\{1,2,3\}$ are intact and the ones in $\{4,5\}$ are befouled.

Now consider the FBQS from Example 2. If f or less servers are faulty, then the set of befouled servers coincides with the set of faulty ones, and all the correct servers are intact. If more than f servers are faulty, then no intact server exists in the system—*i.e.*, $\mathbf{V}$ is the set of befouled servers.

The property of being a quorum is preserved by deleting DSets.

▶ **Proposition 3** ([13]). *Let U be a quorum in FBQS $\langle \mathbf{V}, \mathbf{Q}\rangle$, let $B \subseteq \mathbf{V}$ be a set of servers, and let $U' = U \setminus B$. If $U' \neq \emptyset$ then U' is a quorum in $\langle \mathbf{V}, \mathbf{Q}\rangle^B$.*

1 **process** server($v \in \mathbf{V}$)
2   **var** voted $\leftarrow \perp \in \{tt, ff\}$;
3   **when received** PROPOSE(a) **from some client**
4   **if** voted $\neq \overline{a}$ **then**
5   voted $\leftarrow a$; **send** VOTE(a) **to every server**;
6   **when exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and received** VOTE(a) **or** ACCEPT(a) **from every** $u \in U$
7   **send** ACCEPT(a) **to every server**;
8   **when exists** $B \in 2^{\mathbf{V}} \setminus \{\emptyset\}$ **such that received** ACCEPT(a) **from every** $u \in B$ **and for every** $q \in \mathbf{Q}(v)$, $q \cap B \neq \emptyset$
9   voted $\leftarrow a$; **send** ACCEPT(a) **to every server**;
10   **when exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and received** ACCEPT(a) **from every** $u \in U$
11   **send** CONFIRM(a) **to every client**;

Figure 1 Protocol for binary federated voting in FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$.

The set of befouled servers coincides with the intersection of every DSet that contains all faulty servers.

▶ **Proposition 4** ([13]). *In an FBQS with quorum intersection, the set of befouled servers is a DSet.*

A set of servers B may prevent progress of a server v if B overlaps every one of v's slices—*i.e.*, $\forall q \in \mathbf{Q}(v).\ q \cap B \neq \emptyset$. We say that such B is *v-blocking*. If B is a v-blocking set of befouled nodes, then v is befouled too. The intact servers enjoy the property that faulty servers cannot befoul them, since the DSet of befouled servers is not v-blocking for any intact v.

3.2 Federated voting

We consider the case of federated voting from [13] where the system agrees upon one of the two statements tt or ff, which are contrary to each other—*i.e.*, $\overline{tt} = ff$ and $\overline{ff} = tt$. Binary federated voting solves consensus over Boolean values. The protocol guarantees the safety properties of *integrity*—*i.e.*, every correct server decides at most one value, and if it decides, the value must have been proposed by some client—and *agreement*—*i.e.*, every correct server must agree on the same value. Although distributed, asynchronous consensus lacks the liveness property of *termination* [6], in the setting of the read/write register that we will introduce in Section 5 we achieve some liveness properties by other means (Section 5.3).

Figure 1 implements the server protocol. A client proposes a statement a by sending PROPOSE(a) messages to every server, and a server decides the statement a when it confirms a, after which the server notifies the clients by sending CONFIRM(a) messages to all of them. The following paragraphs explain the three phases of the protocol: voting, accepting, and confirming.

After receiving a PROPOSE(a) message from some client, a server *votes* for statement a—provided it did not vote for $\overline{a}$ before—when it broadcasts the message VOTE(a) to every server (lines 3–5). For simplicity, we assume that servers send messages to themselves.

A server v *accepts* a statement a iff it determines that either

(i) there exist a quorum U such that $v \in U$ and each member of U either votes for a or accepts a (lines 6–7 of Figure 1), or

(ii) each member of a v-blocking set accepts a (lines 8–9 of Figure 1).

After accepting a statement a, the server broadcasts the message `ACCEPT`(a) to every server.

A quorum U *confirms* a statement a iff every server in U accepts a. A server *confirms* a iff it is in such a quorum. After receiving `ACCEPT`(a) messages from every server in the quorum U, a server broadcasts the message `CONFIRM`(a) to every client (lines 10–11).

Now we comment on the need of the three phases of the protocol. We say that a quorum U *ratifies* a statement a iff every member of U votes for a. A server v *ratifies* a iff v is a member of a quorum U that ratifies a. Ratifying guarantees safety to correct servers, but it can only guarantee liveness to a server v if $\mathbf{Q}(v)$ contains at least one quorum slice comprising only correct servers. A set B of faulty servers can violate this property if B is v-blocking. Ratifying is a sufficient condition to accept the statement, but it is not necessary.

On the other hand, accepting allows a well-behaved server that voted for a wrong statement $\overline{a}$ to later accept a. Accepting guarantees safety to correct servers, but it still yields sub-optimal liveness guarantees, since an intact server may accept some statement that other intact servers could be unable to accept. (see Figure 10 and Section 5.4 of [13] for an example). An intact server needs a way to ensure that every other intact server can eventually accept a before acting on it.

Last, confirming requires ratifying the fact that intact servers accepted some statement, which guarantees agreement. We say the system *agrees* on a statement a iff an intact server confirms a statement a. Once an intact server confirms a, then, eventually, every intact server will confirm a.

Theorem 5 below is our novel safety result, which ensures that any pair of correct servers, either befouled or intact, cannot confirm contradictory statements, except for the pathological situation where all servers choose their slices in such a bad way that no server is intact. This correctness statement is stronger than the one given in the Stellar proposal, which only provided such a guarantee for intact servers (Theorem 9 of [13]).

▶ **Theorem 5.** *Consider the protocol in Figure 1 for federated voting over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ that enjoys quorum intersection. If two correct servers v_1 and v_2 confirm statements a and $\overline{a}$ respectively, then no intact server exists in $\langle \mathbf{V}, \mathbf{Q} \rangle$.*

Correct servers are not guaranteed to enjoy liveness, unless they are intact. Theorem 6 below is the known liveness property for intact servers from [13].

▶ **Theorem 6** ([13])**.** *Consider the protocol in Figure 1 for federated voting over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ that enjoys quorum intersection. If an intact server confirms statement a, then, eventually, every intact server will confirm a.*

4 FBQSs with Fallacious Slices

In a setup phase before running federated voting, servers communicate their choice of trust to each other. We study FBQSs with faulty servers that may lie about their quorum slices. Lying about quorum slices may affect computation, since each server computes a quorum system $\mathcal{Q}$ from the other servers's slices, which is later used in the protocol for federated voting (lines 6 and 10 of Figure 1). We extend the definition of FBQS by considering a family of quorum functions $(\mathbf{Q}_v)_{v \in \mathbf{V}}$ indexed by servers such that $\langle \mathbf{V}, \mathbf{Q}_v \rangle$ is an FBQS for every

$v \in \mathbf{V}$. The indexed family $(\mathbf{Q}_v)_{v \in \mathbf{V}}$ reflects each server's subjective view of the choices of trust of other servers. We say that $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$ is an *FBQS with fallacious slices.*

The notions in Section 3 can be duly adapted to FBQSs with fallacious slices. We say that u's slice q is *known by* v iff $q \in \mathbf{Q}_v(u)$. We say that U is a *quorum known by* v iff U is a quorum in FBQS $\langle \mathbf{V}, \mathbf{Q}_v \rangle$. A set $B \subseteq \mathbf{V}$ is v*-blocking* iff it overlaps every one of v's slices known by v itself—*i.e.*, $\forall q \in \mathbf{Q}_v(v).\ q \cap B \neq \emptyset$.

An FBQS with fallacious slides $\langle \mathbf{V}, (\mathbf{Q}_v)_{u \in \mathbf{V}} \rangle$ satisfies *quorum intersection* iff $\langle \mathbf{V}, \mathbf{Q}_v \rangle$ satisfies quorum intersection for every $v \in \mathbf{V}$. Given a set of servers B, to delete B from $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$, written $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle^B$, means to compute the modified FBQS with fallacious slices $\langle \mathbf{V} \setminus B, (\mathbf{Q}^B_v)_{v \in \mathbf{V} \setminus B} \rangle$, where $\mathbf{Q}^B_v(u) = \{q \setminus B \mid q \in \mathbf{Q}_v(u)\}$ for every $v \in \mathbf{V} \setminus B$. A set B of servers is a DSet iff:

- **(i)** (*quorum intersection despite* B) $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle^B$ enjoys quorum intersection, and
- **(ii)** (*quorum availability despite* B) either $\mathbf{V} \setminus B$ is a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$ known by every server $v \in \mathbf{V}$, or $B = \mathbf{V}$.

A server v is *intact* iff there exists a DSet B containing all faulty servers and such that $v \notin B$. Otherwise, v is *befouled.*

The protocol for binary federated voting over FBQSs with fallacious slices coincides with the one in Figure 1, where in line 8 a server v uses the quorum function $\mathbf{Q}_v$, and where in lines 6 and 10 a server v uses the quorum system $\mathcal{Q}$ that is induced by $\langle \mathbf{V}, \mathbf{Q}_v \rangle$. We say a quorum U known by v *ratifies*, *accepts* or *confirms* a statement a iff U respectively ratifies, accepts or confirms a in FBQS $\langle \mathbf{V}, \mathbf{Q}_v \rangle$. A server v *ratifies*, *accepts* or *confirms* a statement a iff v respectively ratifies, accepts or confirms a in FBQS $\langle \mathbf{V}, \mathbf{Q}_v \rangle$.

It turns out that lying about quorum slices may hurt the server who lied, but no others, and the FBQSs with fallacious slices enjoy properties similar to those of the FBQSs of Section 3. In particular, they satisfy the analogous of Theorem 5 and of Proposition 6.

▶ **Theorem 7.** *Consider the protocol for federated voting in Figure 1 over an FBQS with fallacious slices* $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$ *enjoying quorum intersection. If two correct servers* v_1 *and* v_2 *confirm statements* a *and* $\overline{a}$ *respectively, then no intact server exists in* $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$.

▶ **Theorem 8.** *Consider the protocol for federated voting in Figure 1 over an FBQS with fallacious slices* $\langle \mathbf{V}, (\mathbf{Q}_v)_{v \in \mathbf{V}} \rangle$ *enjoying quorum intersection. If an intact server confirms statement* a*, then, eventually, every intact server will confirm* a*.*

In Section 5 we present our read/write register over an FBQS. For simplicity, we will use the plain FBQSs from Section 3.

5 Read/Write Register over an FBQS

To demonstrate that the concept of FBQS is applicable to purposes more general than implementing consensus, we introduce a protocol for a read/write register over an FBQS, which has been inspired by federated voting explained in Section 3.2. Before presenting the read/write register and proving its safety and liveness properties, we comment on the execution model, which resembles those of [12] and [10].

5.1 Execution model and specification

The values that the register stores come from a set $\mathsf{Val} \cup \{\perp\}$. Clients interact with the read/write register by issuing read and write operations.

In order to state properties in the presence of faulty clients, we follow [12] and [10] and allow that a faulty client will stop its execution, at which point it becomes inactive forever. Such a stopping action may be performed for instance by some intrusion detection system that quarantines a process or a machine. The aim of this stopping mechanism is to minimise the effect that faulty clients have on the system. Correct clients could read spurious writes coming from the faulty clients, which would compromise safety for correct clients.

We assume that there is a single object name for the read/write register, and thus we omit it. A *history* is a sequence of invocation and response events, and of stop events. An invocation by a client c is written $\langle c : \mathsf{op}\rangle$, where op is an operation name possibly including arguments, which ranges over $\mathsf{write}(x \in \mathsf{Val})$ and $\mathsf{read}()$. A response to c is written $\langle c : \mathit{rtval}\rangle$ where rtval is the return value—*i.e.*, some $x \in \mathsf{Val}$ in response to a read, an a void value $()$ in response to a write. A response *matches* an invocation if their client names agree. A stop event by client c is written $\langle c : \mathsf{stop}\rangle$, after which client c stops execution.

An *operation* o in a history is a pair consisting of an invocation $\mathsf{inv}(o)$ and the next matching response $\mathsf{resp}(o)$. A history H induces an irreflexive partial order $<_H$ on the operations and stop events in H as follows: $o_1 <_H o_2$ iff $\mathsf{resp}(o_1)$ precedes $\mathsf{inv}(o_2)$ in H; $o_1 <_H \langle c : \mathsf{stop}\rangle$ iff $\mathsf{resp}(o)$ precedes $\langle c : \mathsf{stop}\rangle$; $\langle c : \mathsf{stop}\rangle <_H o_2$ iff $\langle c : \mathsf{stop}\rangle$ precedes $\mathsf{inv}(o_2)$; and $\langle c_1 : \mathsf{stop}\rangle <_H \langle c_2 : \mathsf{stop}\rangle$ iff $\langle c_1 : \mathsf{stop}\rangle$ precedes $\langle c_2 : \mathsf{stop}\rangle$. We say that $<_H$ is the *real-time* order.

A history is *sequential* iff it begins with an invocation, every response is immediately followed by an invocation, or a stop, or no event, and every invocation is followed by an immediate matching response. A *client sub-history* $H|c$ *of a history* H is the sub-sequence of all events in H whose client names are c. A history H is *well-formed* iff for each client c, $H|c$ is sequential. We use $\mathcal{H}$ to denote the set of well-formed histories.

A *sequential specification* for the read/write register is a prefix-closed set of sequential histories. A sequential history H is *legal* iff it belongs to the sequential specification of the read/write register. The sequential specification of our read/write register enforces that a read operation always returns the value written by the last preceding write operation.

A register that meets this sequential specification implements an *atomic register*, since it ensures that for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order [9].

As an intermediate step to proving safety, in our concrete histories we consider writes that come from faulty clients but which may be visible to correct clients. These writes correspond to the *lurking writes* of [10]. We prove that, in the presence of lurking writes, our read/write register is *linearisable* [8, 5] with respect to the specification of an atomic register (Lemma 14 in Section 5.3).

Our main correctness condition is that of *Byzantine fault-tolerant linearisability*[2] (*BFT-linearisability* for short) [10]. BFT-linearisability considers *verifiable histories*, which are histories whose invocation and response events come only from correct clients.

► **Definition 9.** A verifiable history $H \in \mathcal{H}$ is *BFT-linearisable* iff there exists some legal sequential abstract history $H' \in \mathcal{H}$ such that

(i) $H|p = H'|p$, for every $p \in \mathbf{C}_{ok}$,

[2] Our BFT-linearisability has been inspired by the property with the same name in [10], but differs from it in that the number of visible operations after a faulty client is stopped is *finite*, instead of *bounded by a constant.* The strength of our notion of BFT-linearisability lies in between the strengths of Byznearisability from [12] and the original BFT-linearisability from [10] (see Section 7 for a discussion).

(ii) $<_H \subseteq <_{H'}$, and

(iii) for every $c \in \mathbf{C}_{bad}$, if $\langle c : \mathsf{stop}\rangle \in H$ then there exist sub-histories H_1 and H_2 such that $H' = H_1 \langle c : \mathsf{stop}\rangle H_2$ and the number of events by the faulty client c in H_2, this is, $|\{o \in H_2 \mid o = \langle c : \mathsf{op}\rangle\}|$, is finite.

Theorem 15 in Section 5.3 states that our read/write register over an FBQS is BFT-linearisable. Clauses (i) and (ii) of Definition 9 match similar requirements for the correctness notion of linearisability [5]. They ensures that a verifiable history looks plausible to correct clients. Clause (iii) ensures that once a faulty client is stopped, its subsequent effect on the system is limited [10].

We state some liveness properties for the correct operations. In particular, we consider *finite-write termination* [1] (FW-termination for short). A protocol is *FW-terminating* iff the writes always terminate and the reads are guaranteed to terminate unless there are infinitely many writes in the execution. Theorem 16 in Section 5.3 states that correct writes are *wait-free* [7]—*i.e.*, they are guaranteed to terminate—and Theorem 17 in the same section states that the read/write register over an FBQS is FW-terminating after every faulty client has been stopped—*i.e.*, the correct writes terminate, and, after every faulty client has been stopped, either there are infinitely many correct writes in the execution, or the correct reads are guaranteed to terminate.

5.2 Implementation

We assume that the set $\mathbf{C}$ of clients is totally ordered, and that each client $c \in \mathbf{C}$ uses a copy of the same totally ordered set $\mathbf{T}$ of timestamps, and we write $\mathbf{T}_c$ for c's copy of this set. For simplicity, we assume that the set $\mathbf{T}$ of timestamps is unbounded, such that a faulty client cannot exhaust the timestamp space by issuing writes with a very large timestamp. (Practical solutions to this problem when assuming a finite set of timestamps are described in [10].) We let t_0 be a timestamp that is smaller than every $t \in \mathbf{T}$, and let $\mathcal{T} = \{t_0\} \cup \biguplus_{(c \in \mathbf{C})} \mathbf{T}_c$ be the set of global timestamps, which consists of t_0 together with the disjoint union of each client's copy of $\mathbf{T}$. Any timestamp (except t_0) determines uniquely the client that uses it. The set $\mathcal{T}$ of global timestamps is totally ordered where two timestamps $t \in \mathbf{T}_c$ and $t' \in \mathbf{T}_{c'}$ are in lexicographical order when considered as the pairs (t, c) and (t', c').

Clients issue reads and writes, and the protocol runs a round of federated voting for each write. A *write statement* (a *statement*, for short) consists of a pair (x, t) where $x \in \mathsf{Val}$ and t is a timestamp from the set of global timestamps $\mathcal{T}$.

Figure 2 introduces a protocol for a read/write register over an FBQS. Each server contains fields acc and conf—both initially set to $(\perp, t_0)$—which store respectively the statement with the biggest timestamp that was accepted by the server and the statement with the biggest timestamp that was confirmed by the server. Each server also contains the arrays indexed by clients $\mathsf{prop\_client}[c]$ and $\mathsf{conf\_client}[c]$, which store respectively the latest statement proposed by c and the latest statement from c that the server confirmed. If c's proposed and confirmed statements are different, this signals that client c has issued a pending write that the server never confirmed yet. As we will see below, the server uses these fields to determine whether a newly proposed statement is valid before voting for it.

After receiving a `QUERY_A`$(nonce)$ or a `QUERY_C`$(nonce)$ message, a server respectively sends a response `RES_A`$(\mathsf{acc}, nonce)$ or `RES_C`$(\mathsf{conf}, nonce)$ with the accepted statement, or the confirmed statement, respectively, stored at the server (lines 6–9 in Figure 1). If the server never accepted or confirmed anything yet, it sends the statement $(\perp, t_0)$, which we

1 **process** server($v \in \mathbf{V}$)
2 **var** acc $\leftarrow (\bot, t_0) \in \mathsf{Val} \times \mathcal{T}$;
3 **var** conf $\leftarrow (\bot, t_0) \in \mathsf{Val} \times \mathcal{T}$;
4 **var** prop_client$[c \in \mathbf{C}] \leftarrow (\bot, t_0) \in \mathsf{Val} \times \mathcal{T}$;
5 **var** conf_client$[c \in \mathbf{C}] \leftarrow (\bot, t_0) \in \mathsf{Val} \times \mathcal{T}$;
6 **when received** QUERY_A($nonce$) **from** c
7 **send** RES_A(acc, $nonce$) **to** c;
8 **when received** QUERY_C($nonce$) **from** c
9 **send** RES_C(conf, $nonce$) **to** c;
10 **when received** PROPOSE($\langle x, t\rangle_c$) **from** c **and** $t \in \mathbf{T}_c$
11 **if** (conf_client$[c]$ = prop_client$[c] \wedge t >$ prop_client$[c].snd$) **then**
12 prop_client$[c] \leftarrow (x, t)$; **send** VOTE($\langle x, t\rangle_c$) **to every server**;
13 **when exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and received** VOTE($\langle x, t\rangle_c$) **or** ACCEPT($\langle x, t\rangle_c$) **from every** $v' \in U$ **and** $t \in \mathbf{T}_c$
14 **if** $t >$ acc.snd **then** acc $\leftarrow (x, t)$;
15 **send** ACCEPT($\langle x, t\rangle_c$) **to every server**;
16 **when exists** $B \in 2^{\mathbf{V}} \setminus \{\emptyset\}$ **such that received** ACCEPT($\langle x, t\rangle_c$) **from every** $v' \in B$ **and for every** $q \in \mathbf{Q}(v)$, $q \cap B \neq \emptyset$ **and** $t \in \mathbf{T}_c$
17 **if** $t >$ acc.snd **then** acc $\leftarrow (x, t)$;
18 **send** ACCEPT($\langle x, t\rangle_c$) **to every server**;
19 **when exists** $U \in \mathcal{Q}$ **such that** $v \in U$ **and received** ACCEPT($\langle x, t\rangle_c$) **from every** $v' \in U$ **and** $t \in \mathbf{T}_c$
20 **if** $t >$ conf.snd **then** conf $\leftarrow (x, t)$;
21 conf_client$[c] \leftarrow (x, t)$; **send** CONFIRM($\langle x, t\rangle_c$) **to** c;

Figure 2 Protocol for read/write register over FBQS $\langle \mathbf{V}, \mathbf{Q}\rangle$.

call the *init statement*. To avoid replay attacks, the query messages are tagged with a unique *nonce*, that the server sends back in its response. Nonces do not need to be signed.

After receiving a PROPOSE($\langle x, t\rangle_c$) message from client c (lines 10–12) the server first authenticates the signed statement $\langle x, t\rangle_c$ against c's public key, and then it validates the statement by checking that c has no pending writes—*i.e.*, prop_client$[c]$ = conf_client$[c]$)—and that the proposed statement has a bigger timestamp than the last statement proposed by that client—*i.e.*, $t >$ prop_client$[c].snd$. This second condition prevents the server from voting for contradictory statements, and also makes the protocol reliable to duplicated PROPOSE($\langle x, t\rangle_c$) messages, which will be ignored. If the statement is valid, the server updates prop_client$[c]$ and votes for it by broadcasting the message VOTE($\langle x, t\rangle_c$) to every server. The servers repeat the signed statement $\langle x, t\rangle_c$, but they cannot forge spurious statements. Thanks to the conditions of the **if** sentence in lines 11–12, a server will vote for each statement only once.

After receiving either a VOTE($\langle x, t\rangle_c$) or an ACCEPT($\langle x, t\rangle_c$) message from every server in a quorum U such that $v \in U$—or after receiving an ACCEPT($\langle x, t\rangle_c$) message from every server in a v-blocking set B—the server v accepts (x, t) and sends ACCEPT($\langle x, t\rangle_c$) to every server (lines 13-18). If the timestamp t is bigger than that of the accepted statement stored by

1 **function** read() $\in$ Val
2 **pick unique** $nonce$;
3 **repeat**
4 **send** QUERY_C($nonce$) **to every server**; **wait timeout**;
5 **until exist** $U \in \mathcal{Q}$, $x \in$ Val, **and** $t \in \mathcal{T}$ **such that received** RES_C($x, t, nonce$) **from every** $v \in U$;
6 **return** x;

7 **function** write($x \in$ Val)
8 **assume** $x \neq \bot$;
9 **var** $\mathsf{t}, \mathsf{t}_{max} \in \mathcal{T}$;
10 **pick unique** $nonce$;
11 **send** QUERY_A($nonce$) **to every server**;
12 **wait until exists** $U \in \mathcal{Q}$ **such that received** RES_A(_, _, $nonce$) **from every** $v \in U$;
13 $\mathsf{t}_{max} \leftarrow \max\{t \mid$ **received** RES_A(_, t, $nonce$) **from some** $v \in U\}$;
14 $\mathsf{t} \leftarrow \min\{t \mid t \in \mathbf{T}_c \wedge t > \mathsf{t}_{max}\}$ **where** c **is the current client**;
15 **send** PROPOSE($\langle x, \mathsf{t}\rangle_c$) **to every server**;
16 **wait until exists** $U' \in \mathcal{Q}$ **such that received** CONFIRM($\langle x, \mathsf{t}\rangle_c$) **from every** $v \in U'$;

Figure 3 Client's interface for read/write register over FBQS $\langle \mathbf{V}, \mathbf{Q}\rangle$.

the server, then it updates acc with (x, t). Storing the accepted statement with the biggest timestamp is crucial for the safety conditions of the write operation (see Lemma 11 below).

After receiving an ACCEPT($\langle x, t\rangle_c$) message from every server in a quorum U such that $v \in U$ (lines 19–21) the server v updates both conf and conf_client[c], and confirms the statement by sending CONFIRM($\langle x, t\rangle_c$) to c. Storing the confirmed statement with the biggest timestamp is crucial for the safety conditions of the read operation (see Lemma 12 below).

Only a single value can be written on the register for each timestamp. Two statements (x_1, t_1) and (x_2, t_2) such that $t_1 = t_2$ are *contradictory* iff $x_1 \neq x_2$. Since the servers store the current statement proposed by each client (line 4 of Figure 2) and the protocol guarantees that well-behaved servers only vote for each statement once (lines 10–12), it is therefore impossible that well-behaved servers vote for contradictory statements. Therefore, each of the phases of the protocol (voting, accepting, confirming) can be projected into the phases with the same name in federated voting.

▶ **Lemma 10.** *Consider the protocol for read/write register in Figures 2 and 3. For every execution of the protocol and every statement (x, t) that is ever voted in that execution, there exists an execution of binary federated voting on a statement a such that if (x, t) is confirmed and/or accepted in the protocol, then the statement a is respectively confirmed and/or accepted in federated voting.*

Figure 3 depicts the client's interface of our read/write register over $\langle \mathbf{V}, \mathbf{Q}\rangle$. Method read() picks a unique $nonce$ (line 2), and then enters a repeat loop that queries the servers for their confirmed statements (lines 3–5). The loop uses a timeout, and repeats until a quorum U exists such that every server in it returns the same statement (x, t). The loop and the timeout are needed to ensure that intact servers that may still be in the process of confirming some statement have a chance to do so. The read will then return x.

Method $\mathsf{write}(x)$ picks a unique *nonce*, queries the servers for their accepted statements, and waits until a quorum U answers (lines 10–12). The write then picks the maximum timestamp returned by the servers in U, increments it, and assigns $\mathtt{t}$ to it (lines 13–14). Then the write signs the statement $(x, \mathtt{t})$ and sends a $\mathtt{PROPOSE}(\langle x, \mathtt{t}\rangle_c)$ message to every server in the system, thus initiating federated voting on $(x, \mathtt{t})$. The write waits until a quorum U' of servers answers with $\mathtt{CONFIRM}(\langle x, \mathtt{t}\rangle_c)$ (line 17) and returns.

5.3 Correctness

A correct client that invokes $\mathsf{write}(x)$ will initiate federated voting on the statement (x, t) where t is some timestamp. We associate such a statement with its corresponding write operation. From now on we may use 'write' to refer to both the statement and the operation. A faulty client c could single-handedly send some $\mathtt{PROPOSE}(\langle x', t'\rangle_c)$ message and initiate federated voting on some statement (x', t') as well, and we will say that (x', t') is a *faulty write*.

We distinguish the write (correct or faulty) with the biggest timestamp among the ones that have been agreed. We say t is the *current* timestamp iff t is the biggest timestamp of any write that has been agreed. We say v is the *current* value iff (x, t) was agreed and t is the current timestamp. For uniformity, if no statement has been agreed yet we say that t_0 is the current timestamp and $\perp$ is the current value.

We are specially interested in the *visible* writes—*i.e.*, those that could potentially affect a subsequent read. The visible writes include all the correct ones, since these always have a timestamp that is bigger than the current timestamp *at the moment when the associated operation starts* (see Lemma 11 below), and also the faulty writes that have a timestamp bigger than the current timestamp *at the moment when they are agreed.* A correct write (x, t) becomes *visible* when it is agreed. A faulty write (x', t') becomes *visible* when it is agreed iff t' is bigger than the current timestamp at that moment. The visible, faulty writes correspond to the *lurking writes* of [10]. Since lurking writes do not follow the protocol, it is impossible, in general, to ascertain when a lurking write begins and ends [12]. We let a lurking write *start* and *end* instantaneously before and after the moment when it becomes visible. Two visible writes (correct or faulty) *clash* iff one of the writes starts in between the moments when the other starts and becomes visible. Since a correct write (x, t) ends when a quorum U confirms the statement (x, t) then, trivially, every visible write becomes visible in between the moments when it starts and it ends, and two visible writes that are in real-time order do not clash.

Since reads do not alter the abstract state of the register, we need only consider the reads by correct clients (we say *correct* reads for short). We are only interested in the reads that terminate. We distinguish the moment when a terminating read picks up a value. A read *picks up* a write (x, t) when the read receives (x, t) as the confirmed statement from a quorum of servers. Trivially, every correct and terminating read picks up a visible write—or the init statement $(\perp, t_0)$—in between the moments when it starts and ends.

Lemmas 11 and 12 below state useful safety properties of visible writes and correct reads.

▶ **Lemma 11.** *Consider the protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q}\rangle$ enjoying quorum intersection and with some intact server. Let (x, t) be a visible write and let t' be the current timestamp at the moment when (x, t) starts. Then $t > t'$.*

▶ **Lemma 12.** *Consider the protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q}\rangle$ enjoying quorum intersection and with some intact server. If a correct read r picks up a write (x, t),* *then either (x, t) is the init statement and no intact server ever confirmed any write by the time that r picks (x, t) up, or otherwise (x, t) became visible before r picked it up.*

Given a verifiable history $H \in \mathcal{H}$, we construct a sequential abstract history H' which help us to prove Clauses (i)–(iii) in Definition 9, thus proving that our protocol is BFT-linearisable. In an intermediate step, we extend H by inserting every lurking write that is seen by correct clients. For each lurking write (x, t), we insert a pair of consecutive invocation and response events in H at the point where the write (x, t) becomes visible. We call the history H_{ex} so obtained an *extended history*.

Our next step is to prove that an extended history H_{ex} is linearisable with respect to the specification of an atomic register. Operator *seq* defined below takes an extended history and turns it into a sequential one.

▶ **Definition 13.** Let $H_{ex} \in \mathcal{H}$ be an extended history. The history $seq(H_{ex})$ is the sequential history that is constructed recursively as follows:

(i) If H_{ex} does not contain any writes, let $seq(H_{ex})$ contain each read operation from H_{ex} in the same order as the reads pick up the current statement (x, t). Insert in $seq(H_{ex})$ each stop event from H_{ex} before the invocation of the operation that succeeded the stop event in the original history H_{ex}—*i.e.*, as late as possible while preserving $<_{H_{ex}}$.

(ii) Otherwise, let (x, t) be the last write in H_{ex} that becomes visible. Let W^+ be the subset of writes in H_{ex} that clash with (x, t) and that have a timestamp bigger than t. (Notice that W^+ would be empty if no write clashes (x, t), or if all the clashing writes have a timestamp less than t.) Assume that (x', t')—not necessarily different from (x, t)—is the write in $W^+ \cup \{(x, t)\}$ with the maximum timestamp. Let R contain the reads in H that pick (x', t') up. Let S contain the stop events in H_{ex} that do not happen before any operation in $R \cup W^+ \cup \{(x, t)\}$. Construct $seq(H_{ex} \setminus (S \cup R \cup W^+ \cup \{(x, t)\}))$ recursively, and append to it in timestamp order a write operation for each write (x'', t'') in $W^+ \cup \{(x, t)\}$. Append a read operation for each read in R in the same order as they pick (x', t') up. Insert in $seq(H_{ex})$ each stop event from S before the invocation of the operation that succeeded the stop event in the original history H_{ex}—*i.e.*, as late as possible while preserving $<_{H_{ex}}$.

For any extended history H_{ex}, the operator *seq* delivers a linearisation of H_{ex}.

▶ **Lemma 14.** *Let $H_{ex} \in \mathcal{H}$ be an extended history that contains every lurking write that is seen by correct clients. Then, $seq(H_{ex})$ is a linearisation of H_{ex} with respect to the sequential specification of an atomic register—*i.e.*, $seq(H_{ex})$ is a legal history that respects $<_{H_{ex}}$.*

Our main safety result is that the read/write register over $\langle \mathbf{V}, \mathbf{Q} \rangle$ is BFT-linearisable.

▶ **Theorem 15.** *The protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ enjoying quorum intersection and with some intact server is BFT-linearisable.*

We now present our liveness results. As an intermediate step to prove FW-termination we show that correct writes always terminate.

▶ **Theorem 16.** *Consider the protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ enjoying quorum intersection and with some intact server. Then, every correct write terminates.*

The read operation queries the servers for their confirmed statements, and uses a timeout to repeat the query and to give the opportunity for an intact server to confirm the current statement, in case the server did not confirm the current statement yet. An infinite series of consecutive visible writes that are concurrent with the read operation could become visible and preempt termination of the read. However, correct reads are guaranteed to terminate if we assume that the history contains finitely many visible writes.

▶ **Theorem 17.** *Consider the protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ enjoying quorum intersection and with some intact server. If every faulty server has been stopped, then the protocol is FW-terminating*—i.e., *every correct write terminates, and moreover, either every correct read terminates, or the history contains infinitely many correct writes.*

6 FBQSs and Byzantine Quorum Systems

In this section we state the relation between the FBQSs and the classical Byzantine quorum systems from [11]. Together with a quorum system $\mathcal{Q}$, in [11] they consider a *fail-prone system* $\mathcal{B}$, which is a non-empty set of subsets of $\mathbf{V}$ such that none of its elements is contained in another, and some $B \in \mathcal{B}$ contains all the faulty servers. A fail-prone system characterises the failure scenarios that can occur. In [11] they present three variants of Byzantine quorum systems, which are characterised by the properties that $\mathcal{Q}$ and $\mathcal{B}$ satisfy. We focus on the *dissemination quorum systems* of Section 5 of [11]. A quorum system $\mathcal{Q}$ is a *dissemination quorum system* (*DQS* for short) with respect to a fail-prone system $\mathcal{B}$ iff the following properties hold:

(i) (*D-consistency*) $\forall U_1, U_2 \in \mathcal{Q}.\ \forall B \in \mathcal{B}.\ U_1 \cap U_2 \not\subseteq B$, and

(ii) (*D-availability*) $\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q}.\ B \cap Q = \emptyset$.

These two properties resemble the properties of DSets in an FBQS, namely *quorum intersection despite any DSet* and *quorum availability despite any DSet*. Theorem 18 below formalises the connection between FBQSs and DQSs.

▶ **Theorem 18.** *Let $\langle \mathbf{V}, \mathbf{Q} \rangle$ be an FBQS enjoying quorum intersection and such that some intact server exists. Let $\mathcal{D}$ be the set of its DSets. Then, the quorum system $\mathcal{Q}$ induced by $\langle \mathbf{V}, \mathbf{Q} \rangle$ is a DQS with respect to any set $\mathcal{B} \neq \{\mathbf{V}\}$ that is a subset of $\mathcal{D}$ and such that none of $\mathcal{B}$'s elements is a subset of another, and that some $B \in \mathcal{B}$ contains all the befouled servers.*

Theorem 18 defines a one-to-many correspondence between an FBQS and a DQS, where the quorum system $\mathcal{Q}$ is uniquely determined by $\mathbf{Q}$, and the fail-prone system has to be fixed from the subsets $\mathcal{B}$ of $\mathcal{D} \setminus \mathbf{V}$ that satisfy the conditions of the theorem. Such sets $\mathcal{B}$ are indeed fail-prone systems since they contain some element B that contains all the befouled servers, which implies that B contains all the faulty servers. We say that such a fail-prone system is *compatible with* the FBQS. A DQS provides more information than an FBQS (in particular, the choice of fail-prone system). On the other hand, an FBQS generalises a quorum system $\mathcal{Q}$ that is a DQS with respect to the range of fail-prone systems $\mathcal{B}$ that are compatible with the FBQS, and makes the fail-prone system opaque to the client's interface.

Consider the FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ from Example 1. The fail-prone systems $\mathcal{B}_1 = \{\{4, 5\}\}$, $\mathcal{B}_2 = \{\{3, 4, 5\}\}$, and $\mathcal{B}_3 = \{\{3\}, \{4, 5\}\}$ are compatible with $\langle \mathbf{V}, \mathbf{Q} \rangle$. Arguably, the most expressive of them is $\mathcal{B}_3$, which has been picked using the following rule of thumb: pick the smallest elements in the set-inclusion order of $\wp(\mathcal{D} \setminus \{\mathbf{V}\})$—*i.e.*, $\{3\}$ and $\{4, 5\}$—whose union gives the union of the maximal elements in $\wp(\mathcal{D} \setminus \{\mathbf{V}\})$—*i.e.*, $\{3, 4, 5\}$.

The correspondence stated by Theorem 18 warrants that any existing protocol for DQSs can be run on a FBQS, by fixing a fail-prone system that is compatible with the FBQS.

7 Related Work

In [2] they explore realistic modelling for distributing trust on the internet, and they propose general failure patterns for Byzantine fault-tolerant systems that go beyond threshold models. Their *generalised adversary structures* resemble the fail-prone systems of [11] and the DSets of [13] and ours.

Our read/write register over an FBQS addresses faulty clients and allows servers to choose their trust sets independently. The protocol in [10] works with faulty clients, but it uses $3f+1$ servers, with f the threshold for fault-tolerance, and the choice of trust is fixed to any set of $2f+1$ servers. The protocol in Section 6 of [11] also supports faulty clients, but it does so by resorting to the variant of BQSs in Section 4 of [11] called *masking quorum systems*, whose axioms are stronger than those of the DQSs, which are similar in strength to FBQSs's axioms. Furthermore, in an FBQS the failure scenarios emerge from the choices of trust of each server, and therefore they are *opaque* to the protocol, in contrast with the solution in Section 6 of [11], which requires the protocol to be aware of the fail-prone system.

In [12], they assume that faulty clients could leak private keys, and their correctness condition of Byznearisability requires that the number of faulty operations that are seen by correct clients after *all the faulty clients* have been stopped is finite. On the other hand, in [10] they assume the use of cryptographic coprocessors that allow signing without exposing the private key, and their correctness condition of BFT-linearisability is stronger in that it requires that the number of operations from *a faulty client* c that are seen by correct clients after c *has been stopped* is bounded by a constant. Our correctness condition in Section 9 builds upon those in [12] and [10]. As in [10], we assume that faulty clients do not leak private keys, but we only require that the number of visible operations from a faulty client c after it has been stopped is finite. The strength of our BFT-linearisability lies in between Byznearisability from [12] and the original BFT-linearisability from [10].

8 Conclusions and Future Work

In this paper, we have rigorously studied the theoretical foundations of the federated voting protocol in Stellar. In particular, we proved a correctness statement for correct servers, which strengthens the one given in the Stellar proposal that only applies to intact servers. Our correctness statement additionally allows for faulty servers to lie to others about their choice of trust. Furthermore, our read/write register shows how federated voting can be used to solve problems beyond consensus. We have also connected the FBQSs to the well-studied DQSs, which opens up the possibility of running DQS's protocols on top of FBQSs.

The correctness of most constructions on top of FBQSs rely on basic properties of FBQSs that also hold with fallacious slices. It would be routine to implement a read/write register over an FBQS with fallacious slices and prove its correctness as stated by Theorems 15–17.

In Section 18 we explore the relation between FBQSs and the DQSs of [12], and we provide a one-to-many correspondence between an FBQS and a DQS. We believe a correspondence in the other direction (between a DQS and an FBQS) can be defined. Such a correspondence would first consider, for each server v, one slice for each quorum $U \in \mathcal{Q}$ such that $v \in U$. Alas, this straightforward correspondence does not preserve the failure scenarios captured by the fail-prone system $\mathcal{B}$, since $\mathcal{B}$ might not be compatible with the resulting FBQS. In order to preserve the failure scenarios, the information provided by $\mathcal{B}$ should be used to trim each of the servers's slices until the resulting set of DSets contains every element of $\mathcal{B}$. A two-way correspondence between FBQSs and DQSs may help in transferring lower bounds on the number of rounds in register emulations [1], which we leave as future work.

References

1. Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
2. Christian Cachin. Distributing trust on the internet. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, pages 183–192. IEEE Computer Society, 2001.
3. Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017. URL: http://arxiv.org/abs/1707.01873.
4. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
5. Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.
6. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
7. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufman, 2008.
8. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
9. Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.
10. Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *19th International Symposium on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 487–489. Springer-Verlag, 2005.
11. Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
12. Dahlia Malkhi, Michael K. Reiter, and Nancy A. Lynch. A correctness condition for memory shared by Byzantine processes. 1998.
13. David Mazières. The Stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016.
14. David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

A Federated Byzantine Quorum Systems

▶ **Lemma 19.** *Let $\langle \mathbf{V}, \mathbf{Q}\rangle$ be an FBQS enjoying quorum intersection. If an intact server v exists, then every quorum contains some intact server.*

Proof. By Lemma 4, the set of befouled servers is a DSet, and since there is at least one intact server and by quorum availability, then the set of intact servers I is a quorum. Since $\langle \mathbf{V}, \mathbf{Q}\rangle$ enjoys quorum intersection, then for every quorum U, the intersection $U \cap I$, which only contains intact servers, is non-empty. ◀

Proof of Theorem 5. Since v_1 confirmed a, there exists a quorum U_1 that accepts a such that $v_1 \in U_1$. And similarly for v_2, there exists a quorum U_2 that accepts $\overline{a}$ such that v_2 in U_2. Assume towards a contradiction that there exists an intact server v', not necessarily different from v_1 or v_2. By Lemma 19, there is some intact server in U_1 that accepted a, and also there is some intact server in U_2 that accepted $\overline{a}$, but by Theorem 8 in [13] this results in a contradiction and we are done. ◀

B FBQSs with Fallacious Slices

▶ **Lemma 20.** *Let U be a quorum know by v in FBQS with fallacious slices $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$, let $B \subseteq \mathbf{V}$ be a set of servers such that $v \notin B$, and let $U' = U \setminus B$. If $U' \neq \emptyset$ then U' is a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ known by every server.*

Proof. U' being a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ known by every server means that for every $v' \in \mathbf{V} \setminus B$, U' is a quorum in $\langle \mathbf{V} \setminus B, \mathbf{Q}^B_{v'}\rangle$. Since, for every $v' \in \mathbf{V} \setminus B$, both $\langle \mathbf{V}, \mathbf{Q}_{v'}\rangle$ and $\langle \mathbf{V} \setminus B, \mathbf{Q}^B_{v'}\rangle$ are FBQSs, the lemma follows by Theorem 1 in [13]. ◀

▶ **Lemma 21.** *Let $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ be an FBQS with fallacious slices enjoying quorum intersection. If B_1 and B_2 are DSets, then $B = B_1 \cap B_2$ is a DSet, too.*

Proof. Let $U1 = \mathbf{V} \setminus B1$ and $U2 = \mathbf{V} \setminus B2$. If $U_1 = \emptyset$ or $U_2 = \emptyset$ then the lemma follows trivially because $B_1 = \mathbf{V}$ and $B = B_2$, or respectively $B_2 = \mathbf{V}$ and $B = B_1$, and both B_1 and B_2 are DSets. Otherwise, by quorum availability, U_1 and U_2 are quorums in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ known by every server. Since, for any server v, the union of two quorums known by v is a quorum known by v, it follows that $\mathbf{V} \setminus B = U_1 \cup U_2$ is a quorum known by every server, and we have quorum availability despite B.

In order to show quorum intersection despite B, we fix a server $v \in V \setminus B$. Let U_a and U_b be any two quorums known by v in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$. Let $U = U_1 \cup U_2 = U_2 \setminus B$. By quorum intersection of $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$, $U = U_1 \cap U_2 \neq \emptyset$. But then by Lemma 20, $U = U_2 \setminus B$ must be a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$. Now consider that $U_a \setminus B_1$ and $U_a \setminus B_2$ cannot both be empty, or else $U_a \setminus B = U_a$ would be. Hence, by Lemma 20, either $U_a \setminus B_1$ is a quorum in $(\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B)^{B_1} = \langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^{B_1}$, or $U_a \setminus B$ is a quorum in $(\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B)^{B_2} = \langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^{B_2}$, or both. In the former case, note that if $U_a \setminus B_1$ is a quorum in $(U_a \setminus B_1) \cap U = (U_a \setminus B_1) \setminus B_2$, it follows that $U_a \setminus B_2 \neq \emptyset$, making $U_a \setminus B_2$ a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^{B_2}$. By a similar argument, $U_b \setminus B_2$ must be a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^{B_2}$. But then quorum intersection despite B_2 tells us that $(U_a \setminus B_2) \cap (U_b \setminus B_2) \neq \emptyset$, which is only possible if $U_a \cap U_b \neq \emptyset$. ◀

▶ **Lemma 22.** *In an FBQS with fallacious slices enjoying quorum intersection, the set of befouled servers is a DSet.*

Proof. Let $B_{\min}$ be the intersection of every DSet that contains all the faulty servers. It follows from the definition of *intact* that a server v is intact iff $v \notin B_{\min}$. Thus, $B_{\min}$ is precisely the set of befouled servers. By Lemma 21, DSets are closed under intersection, so $B_{\min}$ is a DSet. ◀

▶ **Lemma 23.** *Two intact servers in an FBQS with fallacious slices enjoying quorum intersection cannot ratify contradictory statements.*

Proof. Let B be the set of befouled servers. By Lemma 22, B is a DSet, and by definition $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ enjoys quorum intersection despite B. Assume towards a contradiction that v_1 ratifies a and v_2 ratifies $\overline{a}$. By definition, there must exist a quorum U_1 known by v_1 and containing v_1 that ratified a, and ther must exist a quorum U_2 known by v_2 and containing v_2 that ratified $\overline{a}$. By Lemma 20, since $U \setminus B \neq \emptyset$ and $U_2 \setminus B \neq \emptyset$, both must be quorums in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ respectively known by v_1 and v_2, meaning that v_1 ratified a and v_2 ratified $\overline{a}$ in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$. Since $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ contains only intact servers, all the servers must agree on the choices of quorum slices of each server, and every quorum is known to every server. By quorum intersection despite B, there exists $v \in (U_1 \setminus B) \cap (U_2 \setminus B)$. Such a v must have illegally voted for both a and $\overline{a}$, which contradicts the fact that $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ contains only intact servers. ◀

▶ **Lemma 24.** *The DSet of befouled servers is not v-blocking for any intact v.*

Proof. Let B be the DSet of befouled servers. The statement "for all $v \in V \setminus B$, B is not v-blocking" is equivalent to "for all $v \in V \setminus B$, there exists $q \in Q_v(v)$ such that $q \subseteq V \setminus B$". By the definition of a quorum known by v, the latter holds iff for all $v \in V \setminus B$, $V \setminus B$ is a quorum known by v or $B = V$, which holds by quorum availability despite B. ◀

▶ **Lemma 25.** *Two intact servers in an FBQS with fallacious slices $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ enjoying quorum intersection cannot accept contradictory statements.*

Proof. Let B be the DSet of befouled servers in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ (which exists by Lemma 22). Suppose an intact server accepts statement a. Let v be the first intact server to accept a. At the point v accepts a, only befouled servers in B can claim to accept it. Since by Lemma 24, B cannot be v-blocking, it must be that v accepted a through identifying a quorum U known by v such that every server voted for or accepted a. And since v is the first intact server to accept a, it must mean all servers in $U \setminus B$ voted for a. In other words, v ratified a in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$. Any statement accepted by an intact server in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ will eventually be ratified in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$. Because B is a DSet, $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ enjoys quorum intersection. Because, additionally, B contains all faulty servers, Lemma 23 rules out ratification of contradictory statements. ◀

▶ **Lemma 26.** *Let $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ be an FBQS with fallacious slices enjoying quorum intersection. If an intact server exists, then for every server $v \in \mathbf{V}$, every quorum known by v contains some intact server.*

Proof. By Lemma 22, the set of befouled servers is a DSet, and since there is at least one intact server and by quorum availability, then the set of intact servers I is a quorum known by every server. Since $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ enjoys quorum intersection, then for every quorum U known by any server, the intersection $U \cap I$, which only contains intact servers, is non-empty. ◀

Proof of Theorem 7. Since v_1 confirmed a, there exists a quorum U_1 known by v_1 and such that $v_1 \in U_1$ that accepts a. And similarly for v_2, there exists a quorum U_2 known by v_2 and such that $v_2 \in U_2$ that accepts $\overline{a}$. Assume towards a contradiction that there exists an intact server v', not necessarily different from v_1 or v_2. By Lemma 26 there exists some intact server in U_1 that accepts a, and similarly, there exists some intact server in U_2 that accepts $\overline{a}$. But by Lemma 25 this results in a contradiction and we are done. ◀

▶ **Lemma 27.** *Let B be the set of befouled servers in an FBQS $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle$ with fallacious slices enjoying qourum intersection. Let U be a quorum known to some intact server that contains this intact server, and let S be any set such that $U \subseteq S \subseteq \mathbf{V}$. Let $S^+ = S \setminus B$ be the set of intact servers in S, and let $S^- = (\mathbf{V} \setminus S) \setminus B$ be the set of intact servers not in S. Either $S^- = \emptyset$, or exists a server v in S^- such that S^+ is v-blocking.*

Proof. If S^+ is v-blocking for some $v \in S^-$, then we are done. Otherwise, we show that $S^- = \emptyset$. If S^+ is not v-blocking for any $v \in S^-$, then, by Lemma 24, either $S^- = \emptyset$ or S^- is a quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ known to every server. In the former case we are done, while in the latter we get a contradiction: By Lemma 20, $U \setminus B$ is quorum in $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ known to every server. Since B is a DSet (by Lemma 22), $\langle \mathbf{V}, (\mathbf{Q}_v)_{v\in\mathbf{V}}\rangle^B$ must enjoy quorum intersection, meaning $S^- \cap (U \setminus B) \neq \emptyset$. This is impossible, since $(U \setminus B) \subseteq S$ and $S^- \cap S = \emptyset$. ◀

Proof of Theorem 8. Let B be the DSet of beofuled servers and let $U \not\subseteq B$ be the quorum known by some intact server through which this intact server confirmed a. Let servers in $U \setminus B$ accept a and thus broadcast accept messages. By definition, any server v accepts a if it receives an accept message from every server in a v-blocking set. Hence, the messages sent by the servers in $U \setminus B$ may convince additional servers to accept a. Let these additional servers also broadcast accept messages until a point is reached at which no further servers can accept a. At this point, let S be the servers that accept a (where $U \subseteq S$), let S^+ be the set of intact servers in S, and let S^- be the set of intact servers not in S. S^+ cannot be v-blocking for any server in S^-, or else more servers could come to accept a. By Lemma 27, then $S^- = \emptyset$, meaning every intact server has accepted a. ◀

C Read/Write Register over an FBQS

Proof of Lemma 10. The guards in lines 10, 13, 16 and 19 of the protocol for read/write register in Figure 2 match the corresponding guards in lines 3, 6,8 and 10 of the protocol for federated voting in Figure 1. The additional event handlers in lines 6–9 of Figure 2 correspond to query messages, which do not alter the abstract state of the register. The fields in lines 2–5 of Figure 2 record accepted and confirmed statements by the server, and proposed and confirmed statments from each client that the server heard of. These fields are used to implement the queries's handlers and to enforce that a server never votes for contradictory statements. Each event in a run involving the read/write register can be proected into one event (or none) in a corresponding run of federated voting, and the lemma holds ◀

All the remainig proofs in this appendx implicitly use Lemma 10, which lifts the results about the protocol for federated voting in Section 3 to the protocol of read/write register in Seciton 5.

Proof of Lemma 11. If (x, t) is a lurking write then the lemma holds by definition. Now we show that it holds for correct writes. Since t' is the current timestamp at the moment when (x, t) starts, then either no statement was ever confirmed before and $t' = t_0$, or an intact server confirmed a write with timestamp t', which means that a quorum U accepted a timestamp t' (lines 13–18 of Figure 2). The write operation picks t such that it is bigger than the accepted timestamps queried from a quorum U' (lines 11–14 of Figure 3) before initiating federated voting on (x, t). (Since the query uses a unique *nonce*, there is no confusion between the answers from different queries.) If $t' = t_0$, then by lines 13–14 of Figure 3, t is bigger than t_0, and the lemma holds. Otherwise, by quorum intersection $U \cap U'$ has some intact server v that accepted the timestamp t'. Since an intact server is corrrect by definition, and since a correct servers only update their accepted timestamp with one that is bigger than the one that they store (lines 14 and 17 of Figure 2) then the accepted timestamp stored by v is bigger or equal than t'. Therefore t is also bigger than t'. ◀

Proof of Lemma 12. If r picks up the init statement $(\bot, t_0)$, then r received $(\bot, t_0)$ as the confirmed statement from a quorum U. We show that by that time no intact server ever confirmed any write. Assume towards a contradiction that some intact server confirmed (x', t'). Then, a quorum U' confirmed (x', t'). But this gives a contradiction since by quorum intersection $U \cap U'$ contains some intact server.

Otherwise, a quorum U confirmed statement $(x, t) \neq (\bot, t_0)$. by Lemma 19, U contains some intact server, which confirmed (x, t). If (x, t) comes from a correct client, then it has become visible and the lemma holds. Let t' be the current timestamp at the time when (x, t) was agreed. Some intact server confirmed a write with timestamp t', which means that some quorum U' accepted that write. If (x, t) comes from a faulty client, then by quorum intersection, $U \cap U'$ contains some intact server. Since intact servers are correct, and since correct servers only update their confirmed timestamp with one that is bigger than the one that they store (line 20 of Figure 2), then $t > t'$. Thus, the faulty write (v, t) was agreed at a time when the current timestamp t' was smaller than t, and therefore the faulty write became visible and we are done. ◀

▶ **Lemma 28.** *Consider the protocol in Figures 2 and 3 over an FBQS $\langle \mathbf{V}, \mathbf{Q} \rangle$ enjoying quorum intersection and with some intact server. Let (x, t) and (x', t') be two visible writes with $x \neq x'$. Then $t \neq t'$.*

Proof. Each of (x, t) and (x', t') has been confirmed by some quorum. Since every quorum contains at least one intact server, then, by Lemma 5, (x, t) and (x', t') cannot be contradictory. Therefore, $t \neq t'$ and the lemma holds. ◀

▶ **Lemma 29.** *Let r be a read operation that picks up a write (x, t), and let t' be the current timestamp at the moment when r starts. Then, $t \geq t'$.*

Proof. If $t' = t_0$, then the lemma holds trivially. Otherwise, an intact server confirmed a write with timestamp t' before the read r starts, which means that a quorum U accepted timestamp t'. If r picks up that write, then $t = t'$ and the lemma holds. Otherwise, by Lemma 12, r picks up some write that became visible after the write with timestamp t'—*i.e.*, a quorum U' accepted timestamp t. By quorum intersection, $U \cap U'$ contains some intact server, and since intact servers only update their accepted timestamp with one which is bigger than the one that they store (line 20 of Figure 2), then $t > t'$ and we are done. ◀

Proof of Lemma 14. We proceed by induction on the number of writes in H_{ex}. If H_{ex} does not contain any writes, then the result follows trivially by Definition 13.

Otherwise, assume that (x,t) is the last write that becomes visible in H_{ex}, and let W^+ be the subset of writes in H_{ex} that clash with (x,t) and that have a timestamp bigger than t. Let (x',t') be the write in $W^+ \cup \{(x,t)\}$ with the biggest timestamp, and let R contain the reads in H_{ex} that pick (x',t') up. Let S contain the stop events in H_{ex} that do not happen before any operation $R \cup W^+ \cup \{(x,t)\}$. By the induction hypothesis, $\mathit{seq}(H_{ex} \setminus (S \cup R \cup W^+ \cup \{(x,t)\}))$ is a linearisation of $H_{ex} \setminus (S \cup R \cup W^+ \cup \{(x,t)\})$, and it only remains to show that the operations in $R \cup W^+ \cup \{(x,t)\}$ occur in $\mathit{seq}(H_{ex})$ in a legal order, and that both the operations and the stop events in $S \cup R \cup W^+ \cup \{(x,t)\}$ preserve $<_{H_{ex}}$, both with respect to the other operations in $H_{ex} \setminus (S \cup R \cup W^+ \cup \{(x,t)\})$ and with respect to each other. By Lemmas 11 and 12, and Definition 13, the write (x',t') occurs in $\mathit{seq}(H_{ex})$ after any other other operation with smaller timestamp, and all the reads in R pick (x',t') up and also occur in $\mathit{seq}(H_{ex})$ after (x',t') does. Therefore, all the reads and writes in $R \cup W^+ \cup \{(x,t)\}$ occur in $\mathit{seq}(H_{ex})$ in legal order. By Lemmas 11, 12, 28 and 29, and by Definition 13, the writes in $W^+ \cup \{(x,t)\}$ occur after any other operation in H_{ex} that happens before them in real-time order, and the same is true for the reads in R. That the writes in $W^+ \cup \{(x,t)\}$ and the reads in R preserve $<_{H_{ex}}$ is straightforward, because by Lemma 12 every read that picks up a write does so before the write has become visible. The stop events preserve $<_{H_{ex}}$ by Definition 13, and we are done. ◀

Proof of Theorem 15. Let H_{ex} be an extended history that contains every lurking write that is seen by correct clients, and let H be the verifiable history that contains the correct operations and the stop events in H_{ex}. We show that the verifiable history H and the abstract history $H' = \mathit{seq}(H_{ex})$ meetq Conditions (i)–(iii) of Definition 9. Condition (i) holds since H contain only the correct operations from H_{ex}, and Condition (ii) holds since, trivially, $<_H \subseteq <_{H_{ex}}$ and by Lemma 14. Since a faulty client c can only broadcast a finite number of $\mathtt{PROPOSE}(\langle x,t\rangle_c)$ messages before a stop event $\langle c : \mathsf{stop}\rangle$, and since the statements $\langle x,t\rangle_c$ are signed by c and servers cannot forge them, then the maximum number of operations that could be visible after c is stopped is finite. Therefore, Condition (iii) holds and we are done. ◀

Proof of Theorem 16. Let B be the set of befouled servers in $\langle \mathbf{V}, \mathbf{Q}\rangle$, which is a DSet. By quorum availability despite B, and since some intact server exists, the set of intact server constitute a quorum. Therfore, the query in lines 11–12 of the `write` method in Figure 3 will eventually terminate, and the client will sign the statement (x,t) and initiate federating voting on it. By the use of signatures, servers cannot forge statements, and by Lemmas 10 and 11, servers will never vote contradictory statements. Every intact server will eventually vote for (x,t), and by quorum availability despite B, every intact server v will eventually ratify and accept (x,t) if the server did not previously accept (x,t) trhough a v-blocking set. Thus, every intact server will eventually confirm (x,t) and the method is guaranteed to terminate by quorum availability (line 16 in Figure 3). ◀

Proof of Theorem 17. By Theorem 15, the number of operations from a faulty client c that are seen by correct clients after c has been stopped is finite. In the remainder we prove that a correct read always terminates in the presence of finite visible writes. Let r be a correct read and W be the set of visible writes that are concurrent with r. We show that r terminates and picks up one of the writes in W. We proceed by induction on the size of W. Let $(x,t) \in W$ be the first write that becomes visible. Since (x,t) is visible, some intact server confirmed it, and by Lemma 6 every intact server will eventually confirm it. Let B be the set of befouled nodes in $\langle \mathbf{V}, \mathbf{Q}\rangle$. By quorum availability despite B, the set of intact server is a quorum, and therefore the read's query in lines 3–5 either picks up (x,t) and terminates, or otherwise some other statement gets confirmed by some intact server before this intact node answers the client with the confirmed statement (x, t). In such case, the theorem holds by induction hypothesis on $W \setminus \{(x, t)\}$. If (x, t) is the only statement in W, then the client will eventually pick (x, t) by quorum availability despite B. ◀

D FBQSs and Byzantine Quorum Systems

Let $\langle \mathbf{V}, \mathbf{Q} \rangle$ be an FBQS enjoying quorum intersection and such that some intact server exists. Let $\mathcal{D}$ be the set of its DSets. Then, the quorum system $\mathcal{Q}$ induced by $\langle \mathbf{V}, \mathbf{Q} \rangle$ is a DQS with respect to any set $\mathcal{B} \neq \{\mathbf{V}\}$ that is a subset of $\mathcal{D}$ and such that none of $\mathcal{B}$'s elements is a subset of another, and that some $B \in \mathcal{B}$ contains all the befouled servers.

Proof of Theorem 18. Since no element of $\mathcal{B}$ is a subset of another, and since some element of $\mathcal{B}$ contains all the befouled servers—and thus all the faulty servers—it suffices to show that $\mathcal{Q}$ and $\mathcal{B}$ satisfy D-consistency and D-availability. Let us fix a $B \in \mathcal{B}$. We first prove D-consistency—*i.e.*, $\forall U_1, U_2 \in \mathcal{Q}.\ U_1 \cap U_2 \not\subseteq B$. By Theorem 1 of [13] we know that $U_1 \setminus B$ and $U_2 \setminus B$ are quorums in $\langle \mathbf{V} \setminus B, \mathbf{Q}^B \rangle$. Since $\langle \mathbf{V} \setminus B, \mathbf{Q}^B \rangle$ has quorum intersection, then $(U_1 \setminus B) \cap (U_2 \setminus B) = (U_1 \cap U_2) \setminus B \neq \emptyset$, and therefore $U_1 \cap U_2 \not\subseteq B$. Now we prove D-availability—*i.e.*, $\exists U \in \mathcal{Q}.\ B \cap U = \emptyset$—which holds by letting $U = \mathbf{V} \setminus B$ since $B \neq \mathbf{V}$ and by quorum availability despite B. ◀