A Byzantine Fault-tolerant KV Store for Decentralized PKI and Blockchain

Ryuji Ishiguro

July 17, 2018

https://github.com/yahoo/bftkv

Abstract

Most distributed key-value stores are tolerant to only benign failure, which makes it difficult to run it on the public Internet without additional security measures to protect data integrity from malicious attacks. We developed a key-value store that is tolerant to Byzantine faults, keeping it in mind to run the system over the public Internet. While data integrity is secured among distributed nodes, all transactions are transparent to anyone. The system provides not only a robust kv store but also data secrecy and distributed signing features by a threshold cryptosystem. Integrating encryption and signature schemes with Byzantine fault-tolerant protocols is much more robust than using separated KMS and PKI with an ordinary distributed key-value store, as it maximizes a benefit of distributed systems which fit very well with threshold cryptography. There is no single point of failure in the system. The system by itself is useful as a secure key-value store, but those properties such as Byzantine fault-tolerance, transparency, distributed global storage and threshold cryptosystem can make the system an ideal building block for blockchain technologies.

1 Introduction

How to reach a consensus with Byzantine type failure is the main problem of blockchain technologies. Bitcoin blockchain uses PoW (Proof of Work) with incentives [1]. Majority of public blockchain (or permission-less blockchain) has the same type of consensus mechanism. Another way to establish a consensus is to use BFT (Byzantine Fault-Tolerant) protocols, which is a major mechanism for local/enterprise blockchain (or permissioned system). Some PoS (Proof of Stake) type blockchain technologies use BFT as well to avoid the mining process.

The proposed system is based on the Byzantine quorum system introduced by Malhi and Reiter [2], and a BFT protocol proposed by the same authors [3]. We construct a b-masking quorum system based on the WoT (Web of Trust) graph. We also introduce quorum certificates combined with a threshold authentication scheme to protect data from unauthorized mutation. Unlike a centralized PKI such as X.509, quorum certificates are verified according to a dynamic graph constructed independently at each node. With the strong authentication scheme and flexible certificate mechanism, the system allows anyone to join / leave the network freely without any authorization process.

Another key aspect of blockchain technologies is the global distributed ledger. Bitcoin blockchain uses hash chain – transactions in a specific period are all hashed together with the hash value of the previous block. All bitcoin nodes maintain the same view of the hash chain. The proposed system uses a distributed key-value store with the TOFU (Trust on First Use) policy, that is, the first use of a key locks out others from mutating the value associated with the key. Only the user who has written the key-value first will posses the right to update the value. Also it supports WRITE ONCE permission which guarantees that once a value is written it will never be modified in any way. This special permission will be preferable for applications like the global ledger which must be tamper-proof. The system does not guarantee absolute consistency. Also the order of transactions for different keys does not matter. The system does not even provide eventual consistency among individual nodes. Instead, it has a property such that: $READ(Q_1, x) = READ(Q_2, x), \forall Q_1, Q_2 \in QS$, that is, the consistency is established collectively with a quorum system (QS).

Smart contracts play an important role in some blockchain technologies such as Ethereum and Hyperledger Fabric. As the proposed system is a simple key-value store, it does not provide a platform for smart contracts at the moment. Also, the system itself does not provide any incentive mechanism to discourage nodes to do malicious actions, or to encourage to participate the consensus process. The system has a robust revocation scheme but it does not answer for a question like why would one want to run a node? Although all transactions keep a proof of good or bad actions for each node therefore it will be straightforward to map it to economic incenstives and penalties, we defer fintech discussions to an application layer.

2 Background

This section describes some existing technologies used in the system.

2.1 Quorum Systems

A variety of quorum systems have been used to manage replicated data / storage in distributed systems. We briefly describe the original quorum systems and its extension called Byzantine quorum systems. Later, we construct a byzantine quorum system based on Web of Trust. The system defines two operations, *read* and *write*, between a client and a set of servers called a quorum. A quorum system ($QS \subseteq 2^U$) is a subset of the powerset of all servers (U), and it satisfies a property:

$$\forall Q_1, Q_2 \in QS, Q_1 \cap Q_2 \neq \emptyset \tag{intersection property}$$

With this property, it is guaranteed that the client always retrieves the latest value from at least one server.

Melhi and Reiter [2] extend the property to handle Byzantine failure:

$$\forall Q_1, Q_2 \in QS, \forall B \in BF, Q_1 \cap Q_2 \nsubseteq B$$

where $BF \subseteq 2^U$ and $\cup_{B \in BF} B$ is all Byzantine failure nodes. Especially when $|Q_1 \cap Q_2| \geq 2b + 1$, QS is called a b-masking quorum system. When the client can be dishonet, signed messages are no longer trustworthy therefore we rely on the quorum system to avoid equivocation (aka double spending in blockchain terms).

2.2 Web of Trust

A web of trust is a directed graph $G = (V, E)$, where V is a set of nodes (each of which is a pair of unique ID and public key) and E is a set of trust relationship: when $(v_1, v_2) \in E$, $v1$ *trusts* v_2, i.e., the certificate of $v2$ includes a signature over its public key with the private key of v_1. WoT was introduced by PGP to authenticate certificates of peers without relying on central authorities. We use the same mechanism to authenticate not only end-users' certificates but quorum members' as well.

2.3 Threshold Cryptosystems

Threshold cryptosystems play important roles in this system. We use it not just for fault-tolerance but for improving security.

Shamir's Shared Secret (SSS) is a major tool to construct threshold cryptographic schemes. The system uses SSS for both password authentication and DSA / ECDSA signature schemes. Especially for the latter, the system uses a threshold siganture scheme introduced by Gennaro et al. [4]. For the threshold password authentication, the system adopts SRP [6] with SSS. See 3.3 for the details. SSS uses a $t-1$ degree random polynomial on $\mathbb{Z}_q$

$$f(x) = \sum_{i=0}^{t-1} a_i x^i \bmod q$$

Each *share* is $(i, f(i))_{i=1..n}$. To reconstruct the shared secret $f(0)$, calculate lagrange interpolate from t responses out of n

$$\lambda_j = \prod_{l \in \mathcal{T} \setminus \{j\}} i_l/(i_l - i_j) \bmod q$$

then we get

$$f(0) = \sum_{j \in \mathcal{T}} f(j)\lambda_j$$

where $\mathcal{T}$ is a subset of $\{1..n\}$ and $|\mathcal{T}| = t$.

To construct a (t, n) threshold scheme, we follow the quorum threshold, i.e., $n = |Q|, t = n - b$.

For RSA signatures, it may seem straightforward to apply SSS to the RSA signing process such as:

$$S = M^{f(0)} = \prod_{i \in \mathcal{T}} M^{f(i)\lambda_i} \bmod N$$

however, since to calculate λ_i we need the multiplicative inverse on $\varphi(N)$ which must be kept as secret as the private key, we cannot simply appy SSS to RSA signatures. Shoup, V. [5] solved this problem by getting rid of multiplicative inverse all together but it needs a special construction of the RSA parameters, which makes it difficult to apply existing keys to the method. Therefore we use a simple key hierachy to address this issue. See 3.3 for the details.

For DSA and ECDSA, SSS has preferable properties due to its linearity:

$$f(0) + g(0) = \sum_{i \in \mathcal{T}} (f(i) + g(i))\lambda_i$$
$$f(0) \cdot g(0) = \sum_{i \in \mathcal{T}'} (f(i) \cdot g(i))\lambda_i$$

where

$$f(0) = \sum_{i \in \mathcal{T}} f(i)\lambda_i$$
$$g(0) = \sum_{i \in \mathcal{T}} g(i)\lambda_i$$

T' is a subset of $\{1..n\}$ and $|\mathcal{T}'| = 2t$. These properties make it possible to calculate the DSA / ECDSA signature (r, s), such that:

$$r = g^k \bmod p \bmod q$$
$$s = k^{-1}(x + mr) \bmod q$$

from partial signatures (r_i, s_i), where

$$r_i = g^{k_i} \bmod p \bmod q$$
$$s_i = k_i^{-1}(x_i + mr_i) \bmod q$$

3 Methodology

This section describes quorum systems based on the WoT graph, which are the main building blocks for our BFT protocols. Protocols are always between a client and a quorum. To write a key-value pair, the client issues a request to sign the messages to *Quorum Cliques*, then with the set of signatures it issues another request to *KV quorum* to actually store the key-value:

1. Collect signatures over a message from *Quorum Cliques*

2. Send the message with the collective signature to *KV quorum*

To read a key-value pair, the client issues a *read* request to a *KV quorum*:

1. Collect key-value pairs from a *KV quorum*

2. Choose the latest one

See the section 4 below for details.

In order to complete the system, threshold cryptography based authentication and signature schemes are introduced as well.

3.1 Quorum Cliques

We follow the faulty clients (dishonest writers) scenario described in “Byzantine Quorum Systems” [2, 3], which uses signed messages with b-masking quorum systems. In our system, such quorum system is constructed from maximal cliques in the WoT graph.

If the graph G keeps some cliques and the starting node s has outdegree edges to the cliques within the L distrance, i.e.,

$$\exists (s, u) \in G.E, s.t., u \in QC \text{ and } dist(s, u) \leq L$$

where QC is quorum cliques obtained by the algorithm *GetQC* 7, it will construct a quorum to sign messages. Note that the quorum depends on only the graph G. There are no other configuration data or anything. Such graph is constructed from the quorum certificates.

Quorum Certificates

Quorum certificates represent the proof of trustworthy. Each node keeps its own quorum certificate along with the private key. A quorum certifiate consists of:

- a unique ID

- a public key

- a self signed signature over the ID and public key

- a set of signatures signed by Quorum Cliques

When a node (n_1) is signed by another node (n_2) an edge is added to the WoT graph, i.e., $G.E = G.E \cup \{(n_2, n_1)\}$.

The quorum certificate not only constructs the graph but also gives permissions to clients to mutate the value. Every *write* request includes the client's quorum certificate along with the self signed signature over $\langle x, t, v \rangle$. Each member of QC verifies the signature and the quorum certificate before it sends back the signed message. See algorithms *CheckSigs* 7 and *CheckQuorumCert* 7.

Sybil Attack

When a node is compromised the node can try to make its own cliques with made-up colluding nodes. By the algorithm *GetQC* 7, a node cannot be a member of more than one cliques, which means the compromised node has to sever links to honest nodes itself to make links with the colluding nodes, otherwise the clique can no longer be a member of a quorum.

[graph]

3.2 Key-value Store

Quorum cliques are responsible for signing data collectively. Data is valid only when it has sufficient number of signatures from the cliques.

Once data is signed it can be sent to other nodes that trust the same quorum cliques. Such nodes can form another quorum system and we call it *KV quorum*. The only purpose of this quorum system is to make sure that clients can retrieve the latest key-value. *KV quorum* is typically chosen from $U \setminus QC$ for load balancing.

The client collects $f + 1$ responses and chooses the latest data which has the biggest timestamp. If some servers return an old value or *nil* the client will write back the latest value to those servers. See 4.1 for the actual protocols.

Each member of *KV quorum* must check equivocation and the permission of mutation (TOFU), when it receives the signed transaction.

Equivocation Check / Revocation

Revocation is the only way to keep the system sound in the long run. Even if the quorum system excludes uncertified keys by majority vote, excluding compromised nodes will increase the fault tolerance rate drastically. Each node severs the trust link independently without consulting others when it detects a node signs both $\langle x, t, v \rangle$ and $\langle x, t, v' \rangle$ s.t. $v \neq v'$. Also servers revoke clients when it detects a client signing different values with the same timestamp as well. Once a node is revoked, the node will be excluded from the graph and there is no way to restore it.

Every node in the KV quorum checks if there is a node ($\in QC$) has signed both $\langle x, t, v, s_C \rangle$ and $\langle x, t, v', s'_C \rangle$ where $v \neq v'$. If the node finds such signature in S, it must immidiately revoke the signer. See algorithm *CheckEquivocation* 7.

TOFU Policy

The system enforces the TOFU policy on every write request. If the slot is empty nodes in the KV quorum will simply store the data. If the slot already has data, they first retrieve the latest data and check if the signer is the same as the one of the requested data. See algorithm *CheckTOFU* 7.

3.3 Threshold Password Authentication ($\mathcal{TPA}$)

The quorum system based on the WoT graph guarantees data integrity. The TOFU policy with the quorum certificate prevents unauthorized mutations. The collective signatures make it possible to check equivocation. All those functions rely on the digital signature scheme, which means each node in the system has to have its own key pair for signing and an associated ID. We use a password authentication for:

- recovering from a key-loss situation
- sharing an ID with multiple devices
- data secrecy (roaming encryption)

The system employes a threshold password authentication scheme which is immune from offline dictionary attacks (as long as the number of compromised servers is less than the threshold). We use Shamir's Secret Sharing (SSS) to split the password secret and SRP [6] for the authentication protocol.

To set up the *shares* the client generates a random polynomial for (t, n) SSS on a prime field $\mathbb{Z}_q$, s.t.,

$$f(x) = \sum_{i=0}^{t-1} a_i x^i \bmod q$$

then calculates $n = |Q|$ pairs $(i, f(i)), i = 1..n$. The shared secret is $S = f(0)$. The client also generates a random *salt*. Each *share* will be $\langle i, y_i, v, salt \rangle$, where

$$\begin{aligned} y_i &= f(i) + g^{\pi'} \bmod q \\ v &= g^x \bmod p \\ x &= \pi g^S \bmod q \\ \pi &= \mathrm{OS2I}(h(salt, password)) \bmod p \\ \pi' &= \mathrm{OS2I}(h(password)) \bmod q \end{aligned}$$

p and q are prime numbers such that $p = 2q + 1$ (i.e., p is a safe prime), g is a generator on $\mathbb{Z}_q$. The random polynomial and salt must be generated randomly for each password.

To get a password authenticated

1. The client generates a random number $a \xleftarrow{\mathcal{R}} \mathbb{Z}_q$ and sends

   $$X = g^{a'} \bmod p$$

   where

   $$a' = a - g^{\pi'} \bmod q$$

   to a quorum Q.

2. Each quorum member generates a random number $b_i \xleftarrow{\mathcal{R}} \mathbb{Z}_q$ and calculates a session key

   $$K_i = (Xv^u)^{b_i} \bmod p$$

   then sends back $\{Y_i, B_i, salt, Z_i\}$, where

   $$\begin{aligned} Y_i &= Xg^{y_i} \bmod p, \\ B_i &= kv + g^{b_i} \bmod p, \\ Z_i &= E_{K_i}(P_i, X||B_i). \end{aligned}$$

3. The client collects t responses from the quorum, then calculates

   $$Y_i/g^a = g^{f(i)} \bmod p$$

   and applies the lagrange interpolate

   $$\lambda_j = \prod_{l \in \mathcal{T} \setminus \{j\}} i_l/(i_l - i_j) \bmod q$$

   to $g^{f(i)}$ to get

   $$\prod_{j \in \mathcal{T}} (g^{f(j)})^{\lambda_j} = g^S \bmod p.$$

4. Calculate K_i for each quorum member on the client side, such that

   $$K_i = (B_i - kg^x)^{a'+ux} \bmod p$$

   then decrypts the proof

   $$(P_i, N) = D_{K_i}(Z_i)$$

   and checks if $N = X||B_i$.

Throughout the protocols, k and u represent the SRP parameters, i.e.,

$$\begin{aligned} k &= H(p, g) \\ u &= H(X, B_i) \end{aligned}$$

We use $\{P_i\}_{i \in \mathcal{T}}$ as a proof which will be checked at each server in some protocols.

$\mathcal{TPA}$ for Enrollment

Every node in the system has a unique ID and a public / private key pair associated with the ID. Some nodes can share the same ID but are not likely to share the keys. Each node generates the ID and keys itself then registers it to the quorum cliques to get the public key signed by each member in the cliques. The process requires a password to register the certificate to the system. The password will be checked when the user wants to register multiple devices under the same ID. It also acts as a recovery key when a node has lost the key. As long as the user remembers the password he can generates a brand new key pair with the same ID and register it so that he can update the values associated with the ID.

Enrollment:

1. Generate a unique ID and a key pair.

2. Send the certificate to a quorum with a password to get a quorum certificate.

Key recovery / secondary key generation:

1. Generate a new key pair with the ID.

2. Do the password authentication and get a proof.

3. With the proof send the new certificate to a quorum to get a new quorum certificate.

See 4.3 for the actual protocols.

$\mathcal{TPA}$ for Data Secrecy

The system provides a roaming encryption service. A user encrypts the value at a node then sends the encrypted value to a quorum. He can decrypts the value at any node with the password. The password is stored with $\langle x, t, v \rangle$ in the way of the *threshold password authenticaiton.* The encryption key is $H(\pi g^S)$. S, therefore the random polynomial, has to be randomly generated everytime the data is encrypted at a client.

To decrypt the data the user specifies the variable x along with the password then the client starts the threshold password authentication process first to get the proof. With the proof the client proceeds the *read* process and decrypts the value with the above key.

3.4 Distributed Signing

Some crypto-currency systems still use simple digital signature schemes to sign transactions. If private keys are stolen all assets can be transferred to someone's address without an authorization of the original users. One of the means to mitigate such situation is to use *multi-sig*. With the multisig scheme multiple parties need to be involved in the signning process. Most systems adopt multisig as an add-on security measure to the blockchain technologies. BFTKV natively supports distributed signing schemes based on threshold cryptography, which is compatible with the standard algorithms therefore the verifiers do not need to change any way to process the transactions.

As a PKI platform, protecting CA's private keys must be the most important *feature*. At the same time, certificate signing requests should be processed quickly and automatically. The distributed signing feature satisfies those mandate requirements.

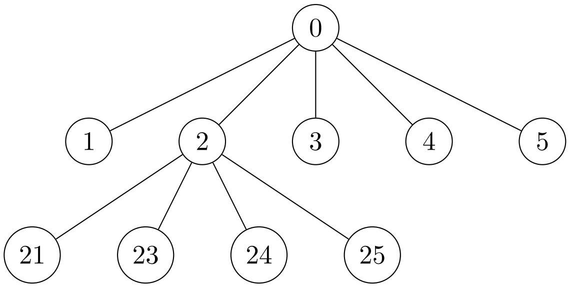
The system takes advantages of the distributed platform to make a signature without revealing the private key. Once the key is distributed among quorum members, the complete private key will never appear at any server or client. The system supports three threshold signature algorithms: RSA (PKCS1.5), DSA and ECDSA. Our scheme takes existing keys as-is and distributes it to multiple servers, which makes it easy to migrate.

RSA

It is straightforward to construct a (n,n) "threshold" scheme on RSA [7, 8]. Decompose the private key d into $\{d_i\}_{i=1..n}$ $s.t., d = \sum_{i=1}^{n} d_i \bmod \varphi(N)$, and the signature is calculated from partial signatures $S_i = M^{d_i} \bmod N$ then

$$S = \prod_{i=1}^{n} S_i = M^d \bmod N$$

To convert the (n,n) threshold scheme to a somewhat (t,n) threshold scheme, we construct the partial keys recursively. In the following tree, if a node (2) is a faulty its key will be compensated by other nodes (1, 3, 4, 5) with the keys (21, 23, 24, 25) respectively, i.e., $S = \prod_{i\in\{1,2,3,4,5\}} S_i \bmod N$ where $S_2 = \prod_{i\in\{21,23,24,25\}} S_i \bmod N$.



The number of keys each node needs to keep will be increased exponentially as $n-k$ becomes big. Up to $(7,10)$ threshold seems practical.

DSA, ECDSA

BFTKV implements a DSS threshold scheme introduced by Gennaro et al. [4]. Since the scheme has a restriction such that $n \geq 2t$ in the (t,n)-threshold scheme, we no longer be able to use the quorum threshold for (t,n). But, we follow the protocols between the client and a quorum, i.e., the client sends a signing request to a quorum (multicast the message to all quorum members), then collect the responses. Our signing protocol consists of three phases:

1. Collect joint shared secrets generated by each quorum member
2. Distribute the secrets to the quorum and calculate $r = g^{k^{-1}} \bmod p \bmod q$ where k is the joint Shamir's shared secret (i.e., $k = \sum k_i$)
3. Distribute r to the quorum and calculate $s = k(m + xr) \bmod q$ from each $s_i = k_i(m + x_i r) \bmod q$ returned from each quorum member

4 Protocols

The system uses Phalanx [3] for the underlying *read/write* operations. To maintain a graph for the quorum system, we extend the protocols with *join/leave*. Also *register* protocol is used to get quorum certificates with the threshold password authentication.

Notations: $\langle x, t, v, s_C, S\rangle$ is an ordered set that denotes the protocol data. s_C and S can be omitted.

x is the variable and an arbitrary length octet string. The variable will be the key on the key-value store.

t is the timestamp and a 64-bit non-negative integer. $2^{64} - 1$ is a special value to denote that the variable is no longer able to be updated.

v is the value and an arbitrary length octet string.

s_C is an object of the signature and quorum certificate of a client C. $s_C.sig$ is the signature over $\langle x, t, v \rangle$ signed by the private key of C. $s_C.cert$ is the quorum certificate of C. $s_C.cert.ID$ is the unique ID to identify each node.

S is an unordered set of the signature object. Each $S_i \in S$ has the same structure of the s_C. The signers must be quorum members.

4.1 Read / Write

read/write protocols are done between a client (C) and a quorum (Q). A quorum is chosen from a quorum system ($QS \subseteq 2^U$) which is a subset of the powerset of all nodes (U). To write a value v into a variable x, we follow the write protocol specified in Phalanx.

[Write]

$$\begin{aligned}
C &: \text{choose a quorum from quorum cliques} \\
&\quad Q \in QC \\
C \rightarrow Q &: \textit{get timestamp}(x) \\
C \leftarrow Q_i &: t_i \\
C &: \text{collect } 2b+1 \text{ timestamps:} \\
&\quad \{t_i\}_{i \in \mathcal{T}}, |\mathcal{T}| = 2b+1, \mathcal{T} \subseteq Q \\
C &: \text{choose the maximum timestamp} \\
&\quad t = max(t_i) + 1 \\
C \rightarrow Q &: \textit{get signature}(\langle x, t, v, s_C \rangle), \\
&\quad \text{where } s_C = (Sign_C(\langle x, t, v \rangle), Cert_C) \\
Q_i &: \text{verify } s_C \text{ with } C\text{'s quorum certificates} \\
Q_i &: \text{check the TOFU policy} \\
C \rightarrow Q_i &: S_i = \{\langle x, t, v, s_C \rangle\}_{Q_i} \\
C &: \text{collect signatures} \\
&\quad S = \{S_i\}_{i \in \mathcal{T}} \\
C &: \text{choose a quorum from } Q' = U \setminus QC \\
C \rightarrow Q' &: \text{write}(\langle x, t, v, s_C, S \rangle) \\
Q'_i &: \text{verify the signature set } S \\
Q'_i &: \text{do the equivocation check} \\
Q'_i &: \text{store } \langle x, t, v, S_C, S \rangle
\end{aligned}$$

[Read]

$$\begin{aligned}
C &: \text{choose a quorum } Q \in U \setminus QC \\
C \rightarrow Q &: \textit{read}(x) \\
C \leftarrow Q_i &: M_i = \langle x, t, v, s_C, S \rangle \\
C &: \text{collect } f+1 \text{ responses} \\
C &: \text{verify the signature set } S \\
C &: \text{do the equivocation check} \\
C &: \text{choose the latest timestamp} \\
&\quad t_L = max(M_i.t) \\
&\quad [\textit{writeback}] \\
C &: Q' = \{Q_i \subset Q | M_i.t < t_L\} \\
C \rightarrow Q' &: \textit{write back}(\langle x, t_L, v, s_C, S \rangle)
\end{aligned}$$

4.2 Join / Leave

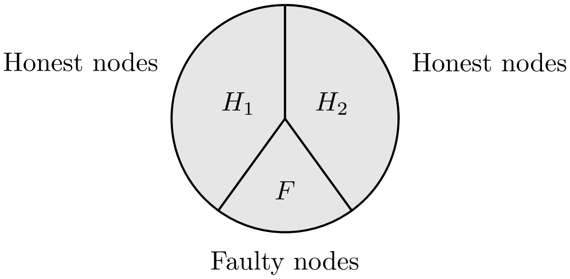
Any node can join / leave anytime by sending its quorum certificate to nodes it trusts. The node received the request verifies the certificate and adds it to the local graph which will be returned to the caller node. To leave the network, a node broadcasts its quorum certificate to the quorum it belongs to. The node that has received the graph constructs its own local graph from the graphs, then sends the join request to nodes that have not been connected. See algorithm *Join* 7.

4.3 Register

Each node generates its own public/private key pair, then sends the public key part to a quorum to get a quorum certificate. Each quorum member keeps all certificate issued to clients along with a partial password secret. If it finds a certificate request that has the same ID as one of the stored certificates it will sign it only if the password authentication is done. The client will get a “proof” of the authentication when it is finished. See algorithm *Register* 7

5 Security Analysis

We look into attacks against the fundamental property: $READ(Q_1, x) = READ(Q_2, x)$ for $\forall Q_1, Q_2 \in QS$, which is known as equivocation. The best that attackers can do is divide a clique into two sets and ask each set to sign $\langle x, t, v \rangle$ and $\langle x, t, v' \rangle$ separately. Then do the *write* protocol for the target nodes with collected signature sets S and S'. Honest servers will refuse the request because it does not satisfy the basic b-masking quorum condition: $|S| \geq b+1$. But with $b+1$ colluding nodes, the attack will succeed.



The maximum number of signatures dishonest clients can get is $b + (n - b)/2$. Therefore, to overcome the attack we need

$$n - b > b + (n - b)/2 \Rightarrow n > 3b$$

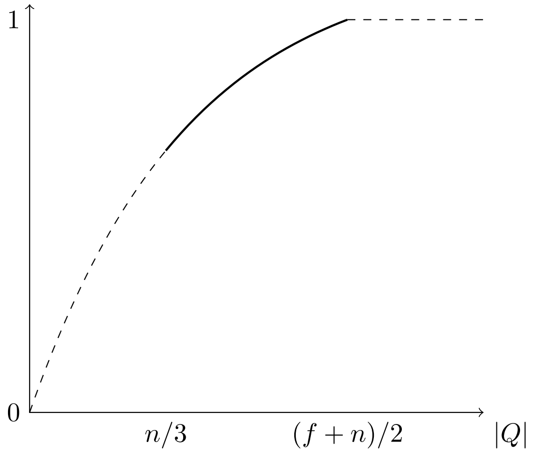
Detecting equivocation on read

Even if the number of faulty nodes exceeds the above threshold, the system can detect malicious actions with the following probability

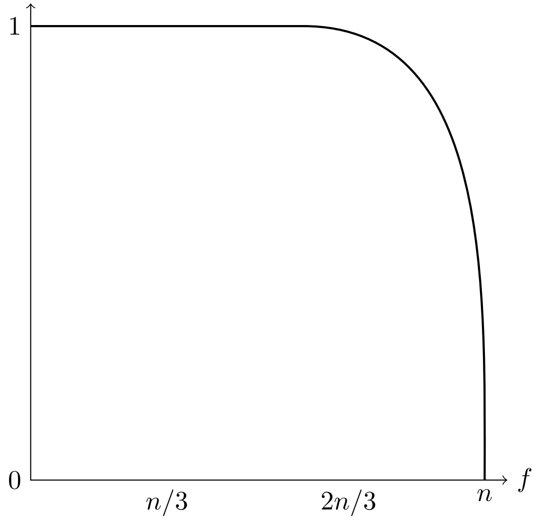
$$\begin{aligned} F_p &= Pr[Q \cap H_1 \neq \emptyset \wedge Q \cap H_2 \neq \emptyset] \\ &= 1 - Pr[Q \subseteq F \cup H_1] \\ &= 1 - ((n + f)/2n)^{|Q|} \end{aligned}$$

when $f > b$ and $|Q| < (f + n)/2$, where f is the number of the faulty nodes, assuming the sizes of H_1 and H_2 are the same.

In the case of $f \leq b$ the detection rate is always 100% because it is guaranteed that clients can always find a valid value and anything other than that is the result of malicious actions. When the number of faulty nodes exceeds the threshold, i.e., $f > b$, it will be possible that the client cannot detect all malicious actions. Since the minimum quorum size is $(n - 1)/3$ it does not need to consider the case where $|Q| < n/3$. Also, if the size exceeds $(f + n)/2$ any quorum always includes at least one node from each H_i which makes the detection rate 100%.



For example, assume we choose a quorum $|Q| = 3b + 1$ out of $n = 4b + 1$, which is the default setup of the kv quorum system, the detection rate is 100% up to $f = 2b$ failure nodes.



6 Applications

We discuss some potential appliations of BFTKV.

6.1 Decentralized PKI with DKMS

User authentication is a long-standing problem for end-to-end systems. Even if we have semantically secure cryptographic protocols to exchange data between users, if it was with a wrong one, the whole security system would not make sense. On the other hand, once we have a robust user authentication scheme, we can build up many kinds of security systems on top of that, such as PGP and Signal. Our goal is to construct an infrastructure to exchange public keys that represent users' identities. Exchanging public keys can be done in person, using a QR code, confirming the fingerprint of public keys, etc. Those methods seem to be relevant for some situations, such as sending money. Also, public key infrastructures using central authorities, such as X.509 which is based on chain of trust, are widely used. A PKI like X.509, however, still have a problem when issuing a certificate to each end user. CAs issue certificates to corporates, organizations, and individuals based on trustworthiness of requesters but for end users whose authenticity is not easy to be proven, we have the same basic issues. From end users' point of view, blindly relying on a central authority based on its authenticity is no longer secure and contradict the end-to-end philosophy.

Our proposed PKI does not "strongly" rely on central authorities, yet it does not require to exchange public keys in person. Here are the high level system requirements:

- Scalability – the system can grow without affecting the current running services
- Transparency – anyone can monitor every system activity
- Quantifiability – security and efficiency can be formally analyzed
- Robustness – the system has to recover from erroneous situations by itself
- Privacy – the system should not reveal unnecessary information about users
- Non-interactivity – a client may not be able to interact peers before sending a message. This particular requirement makes it difficult to design a system that guarantees the "what I saw is what you see" concept. SMTP, for example, is not a mutual explicit authentication protocol. When an email is encrypted then sent out, if it is encrypted with a wrong key, it will be too late – someone in the middle could read the email when the recipient receives the email and notice that the encrypted email is not actually for her.

6.2 Consensus Mechanism for Blockchain Technologies

Key-value stores can be suitable to store transactions. Without knowing the key it will be difficult to access the value – in this sense it can be less transparent than other ledgers such as hash chain. To keep the transactions we can simply use the hash value of the whole transaction data. As every transaction should have a public key ("address") and transaction ID in the sense of bitcoin Tx, the hash value should be unique. We write the value with the WRITE ONCE permission.

7 Conclusions and Future Work

We developed an efficient and secure key-value store. The graph driven architecture gives flexibility to the system, which makes it possible to join / leave freely without any additional authorization mechanism. Unlike other key-value stores, BFTKV provides strong encryption and signature schemes with a threshold cryptosystem as well. It also solves nasty problems like recovering from a key-loss situation and supporting multiple devices under the same user ID.

Scalability of bitcoin blockchain is excellent – you can add as many nodes as you want and it will not affect the whole system, but the latency and throughput of the whole process is terrible – it takes one hour to settle a transaction. On the other hand, BFT type of blockchain settles transactions in a range from sub milliseconds to a few seconds, but scaling out is difficult. Specifically, with quorum systems we have theoretical lower bounds such as $n = 3f + 1$ and we cannot do anything about it.

BFTKV can separate the role of the node into two parts: signing and reading / writing, corresponding to the auth quorum and kv quorum for load balancing. Once a transaction gets a collective signature, the transaction can be stored in any way. The only concern is how to know a transaction is latest. Each transaction has the timestamp and we can easily get the latest $\langle x, t, v \rangle$ corresponding to a x in a node but we do not know if other nodes might have a newer value. BFTKV addresses this issue by a simple quorum system, i.e., as long as $\forall Q_1, Q_2 \in QS, |Q_1 \cap Q_2| > f$ holds, at least one node in a quorum has the latest value and the client will choose that one. This method is simple but it is difficult to scale out. We need to collect data from at least $f + 1$ nodes, which depends on the total number of servers.

Another concern is the number of collective signatures. Increasing the number of servers does not help improve the throughput at all, because every available server in quorum cliques needs to get invovled in the signing process. Also it will increase the number of the collective signatures so that each client and kv node needs to verify more signatures. One of the ideas to address this issue is to use a threshold signature to combine the set of collective signatures so that each node verifies the signature only once. But we need a dealer for the threshold signature scheme and it will contradict the decentralized concept and make the system less flexible.

References

[1] Nakamoto, S. (2008) Bitcoin: A Peer-to-Peer Electronic Cash System

[2] Malhi, D. and Reiter, M. (1998). Byzantine Quorum Systems

[3] Malhi, D. and Reiter, M. (1998). Secure and Scalable Replication in Phalanx

[4] Gennaro, R., Jarecki, S., Krawczyk, H. and Rabin, T. (1999). Robust Threshold DSS Signatures

[5] Shoup, V. Practical Threshold Signatures

[6] Wu, T. (2000). The SRP Authentication and Key Exchange System (RFC 2945)

[7] Garay, J. A., Gennarob, R., Jutlab, C., Rabin, T. (2000) Secure distributed storage and retrieval

[8] Rabin, T. (1998) A Simplified Approach to Threshold and Proactive RSA

Acknowledgements

We would like to thank Idit Keidar for the important suggestions about the quorum system. In fact, the idea of separating the KV quorum system from the auth quorum system first appears in her email messages. Also Edward Bortnikov and Juan A. Garay helped us move the project forward in many ways.

Algorithms

Algorithm 1: GetQC

```
Input: $G$: a graph, $s$: the start node, $L$: the
       maximum distance
$queue \leftarrow (s, 0)$;
$QC = \{\}$;
repeat
    $(v, d) \leftarrow dequeue()$;
    if $d \geq L$ then
        break
    end
    $QC = QC \cup$ FindMaximalClique($v$);
    for each node $n \in v.adj$ do
        enqueue($n, d + 1$) if $n$ has not been
          visited;
    end
until $queue = \{\}$;
Check if $\forall C_1, C_2 \in QC, C_1 \cap C_2 = \emptyset$;
return $QC$
```

Algorithm 2: FindMaximalClique

```
Input: $G$: a graph, $s$: the start node
$C \leftarrow s$;
for $v \in G.V$ do
    $C = C + v$ if $(c_i, v)$ and $(v, c_i)$ for $\forall c_i \in C$;
end
if $|C| < 4$ then
    return $\perp$
end
return $C$
```

Algorithm 3: CheckSigs

```
Input: $S = \{S_i\}, S_i = Sign_{Q_i}(\langle x, t, v, s_C \rangle)$
$Cliques = GetQC()$;
for $clique \in Cliques$ do
    if not $VerifySS(clique, S)$ then
        return $False$;
    end
end
return $True$;
```

Algorithm 4: CheckQuorumCert

```
Input: Cert
$Clique = FindMaximalClique(self)$;
$Counter = 0$;
for each $c \in Cert$ do
    if $c.Issuer \in Clique$ and
     $Verify(c.Issuer, c.Signature)$ then
        $Counter{+}{+}$;
    end
end
if $Counter > 2 \cdot |Clique|/3$ then
    return $True$;
else
    return $False$;
end
```

Algorithm 5: Equivocation Check

```
Input: $req = \langle x, t, v, s_C, S \rangle$
$z = Store[x, t]$;
if $z \neq \perp$ and $req.v \neq z.v$ then
    $Revoke(req.S \cap z.S)$;
end
```

```
Algorithm 6: TOFU enforcement
  Input: $req = \langle x, t, v, s_C, S \rangle$
  Verify $req.s_C$ with quorum certificate;
  if $Store[x, 0] = \bot$;
   then
      $Store[x, t] = req$;
  else
      $last = Store[x, t-1]$;
      if $last.s_C.cert.ID = req.s_C.cert.ID$;
       then
          $Store[x, t] = req$;
      else
          Error;
      end
  end
```

```
Algorithm 7: Join
  Input: Cert
  $G.V = Cert.sigs[*].cert$;
  $peers = G.V$;
  for $peers = \bot$ do
      $newPeers = \{\}$;
      for $peer \in peers$ do
          Send $Cert$ to $peer.addr$;
          $certs$ = Receive();
          $newPeers = newPeers \cup (certs \setminus G.V)$;
          $G.V = G.V \cup certs$;
      end
      $peers = newPeers$;
  end
```

```
Algorithm 8: Register
  Input: $req$: a client certificate, $proof$: the
         proof of the password authentication
  $found = Store[req.x]$;
  if $found \neq \bot$ and
   $req.s_C.ID = found.s_C.ID$ then
      $clique = FindMaximalClique(self)$;
      if $proof \subseteq clique$ and
        $|proof| \geq |clique| \cdot 2/3$ then
          $Sign(req)$;
      end
  end
```