

A BFT KV Storage

Yahoo Paranoids
<https://github.com/yahoo/bftkv>

Introduction

We design a distributed BFT key-value storage based on Byzantine quorum systems¹. The system consists of:

- Phalanx² -- a BFT protocol designed by the same authors who introduced Byzantine quorum systems. We use the protocol for the underlying READ/WRITE operations.
- Web of Trust -- to construct a b -masking quorum system that satisfies the Byzantine quorum properties.
- Quorum certificate -- to prevent unauthorized mutation.

Unlike consensus protocols, the proposed system does not guarantee that all replica nodes have the same data. Instead, it has a property: $READ(Q_1, x) = READ(Q_2, x), \forall Q_1, Q_2 \in QS$. Consensus is established collectively. Key-value store is useful by itself but we also show how collective consensus can be used to construct global shared ledgers.

Background

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:

$$\forall Q_1, Q_2 \in QS, Q_1 \cap Q_2 \neq \emptyset \quad (\text{intersection})$$

The write operation is done between a client (c) and a quorum (Q), writing a value (v) to a variable (x):

Choose $Q \in QS$, and for each $q \in Q$ do: write($x, <v, t>$)

The read operation is as follows:

¹ D.Malhi, M.Reiter. "Byzantine Quorum Systems"
[<http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/papers/bquorum-dc.pdf>]

² D.Malhi, M.Reiter. "Secure and Scalable Replication in Phalanx" [<https://www.cs.unc.edu/~reiter/papers/1998/SRDS-1.pdf>]

1. Choose $Q \in QS$, and for each $q \in Q$ do: $\langle v, t \rangle_q \leftarrow \text{read}(x)$
2. Choose the pair that has the latest timestamp: $\langle v, t \rangle$
3. Write the value back to Q : for each $q \in Q$ do: $\text{write}(x, \langle v, t \rangle)$

Note Q can be chosen independently of the write operation. Because of the intersection property at least one server should have the right value.

The original quorum system handles only the benign failure (fail stop) case. Malkhi and Reiter [ref] extend the system so that it can deal with byzantine failure, calling it Byzantine quorum systems. With the signed message (SM) byzantine, the intersection property is extended as:

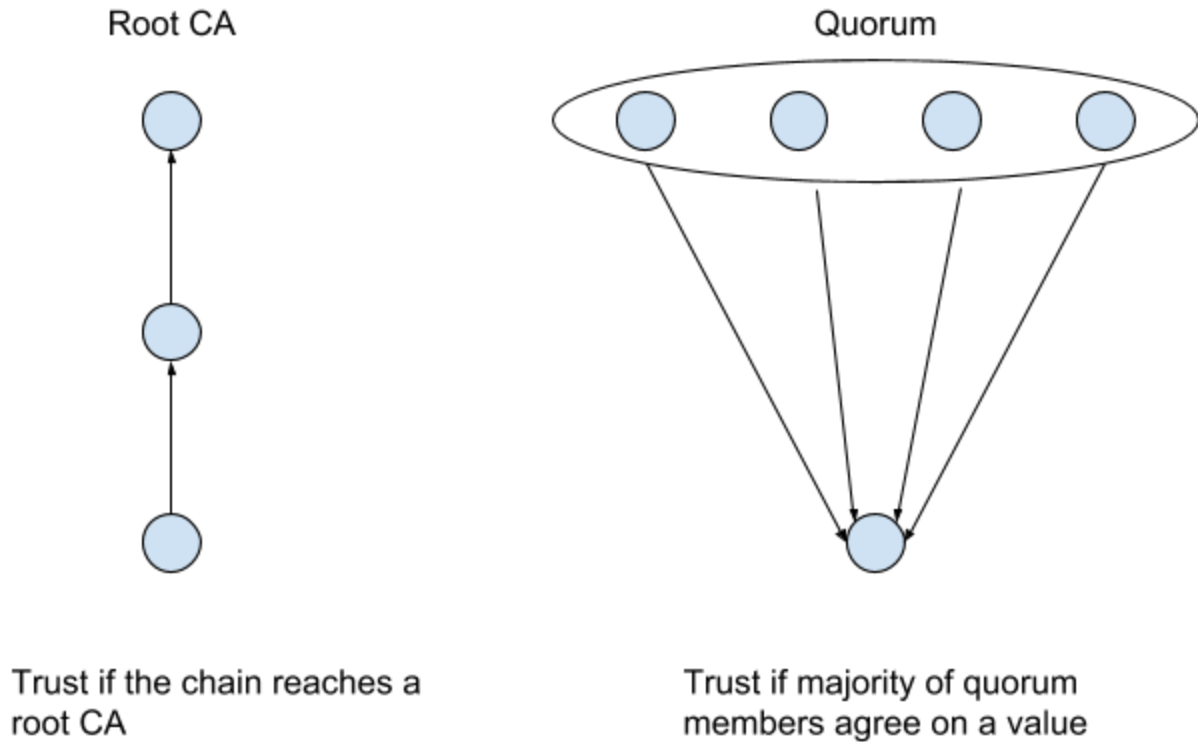
$$\forall Q_1, Q_2 \in QS, \forall B \in BF : Q_1 \cap Q_2 \subseteq B$$

Where BF is a subset of the powerset of U and $\bigcup_{B \in BF} B$ is all byzantine failure nodes.

Quorum Certificate

Practical Byzantine Fault Tolerance system (Castro [ref]) uses a byzantine quorum system to order requests at each node to replicate data among them. A node, either client or server, writes a value to a quorum and if it collects at least $f+1$, where f is the maximum number of replicas that may be faulty, positive responses, the process is successful and the value is considered to be verified.

Quorum certificates are not a data structure such as X.509 but more like a protocol to certify a value stored in a quorum:



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 $(v_1, v_2) \in E$, v_1 trusts v_2 , i.e., the certificate of v_2 includes a signature over its public key with the private key of v_1 . WoT was introduced by PGP [ref] 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.

System Design

The system uses two quorum systems: one is to collect signatures, called authority (QS_A), and another one is to read/write key-value pairs (QS_B). The protocols do not depend on quorum systems -- any quorum system can plugged-in.

Abstract Protocols

$WRITE(x, v)$ requests to store a value v to a variable x .

[write]

C : choose a quorum $Q \in QS_A$, s.t. $|Q| \geq 3b + 1$

$C \rightarrow Q$: “get time”, with the variable x
 $C \leftarrow Q$: collect timestamps $\{t_1, t_2, \dots, t_n\}$, $n \geq 2b + 1$
 C : $t \leftarrow \max(t_i) + 1$
 $C \rightarrow Q$: “echo request”, $\langle x, t, v, sig \rangle$ where $sig = \text{Sign}(\text{Priv}_C, \langle x, t, v \rangle)$
 $q_i \in Q$: check the signature with the C ’s certificate
 verify C ’s certificate with a quorum certificate if x is not “first use”
 $s_i \leftarrow \text{Sign}(\text{Priv}_{q_i}, \langle x, t, v, sig \rangle)$
 $C \leftarrow Q$: collect signatures $S = \{s_1, s_2, \dots, s_m\}$, $m > b + (n - b)/2$
 C : choose a quorum $Q' \in QS_B$
 $C \rightarrow Q'$: “write”, $\langle x, t, v, sig, S \rangle$
 $q_i \in Q'$: verify sig
 check if the number of valid signature in S is $> b + (n - b)/2$
 then check if t has not been written into x
 store $\langle x, t, v, sig, S \rangle$ into the storage
 $C \leftarrow Q'$: ack / nack
 C : if the number of acks is $\geq 2f + 1$ then success, otherwise fail

$READ(x)$ requests to retrieve the latest value of variable x .

[read]

C : choose a quorum $Q \in QS_B$
 $C \rightarrow Q$: “read”, x
 $C \leftarrow Q$: collect key-value pairs $\{\langle x, t_i, v_i, sig_i, S_i \rangle\}$ up to $2f + 1$
 C : revoke all signers who have signed $\langle t, v \rangle$ and $\langle t, v' \rangle$
 discard pairs that do not have sufficient number of valid signatures
 choose a pair $\langle t, v \rangle$ s.t. $\#(\langle t_i, v_i \rangle) \geq b + 1$ and has maximum timestamp

[write back (optional)]

C : $Q = \{ \text{all nodes that have not returned a pair } \langle x, t_i, v_i \rangle \}$
 $C \rightarrow Q'$: “write back”, $\langle x, t_i, v_i, sig, S \rangle$
 $q_i \in Q'$: same as “write”
 // no response

- On “echo request”, if a variable x already has a value, each quorum member must verify that the new value has a signature signed by the same client or a quorum trusts the client cert: $(\text{Cert}_{prev} = \text{Cert} \vee QV(Q, \text{Cert})) \wedge \text{verify}(\text{Cert}, \{x, t, v\}_C) = \text{true}$ ³, where $QV(Q, \bullet)$ is to verify the cert with quorum certificates.
- Q' can be larger than Q in the “write” protocol, which means a small set of signatures can be propagated to a large set of servers, therefore the burden of verifying signatures can be kept reasonable.

³ This is a TOFU policy. To mitigate the key loss situation, clients will be encouraged to have more than one keys stored in different devices.

- On the “read” protocol, if the client finds $\langle x, t, v \rangle$ and $\langle x, t, v' \rangle$ with the same signer, either a client or a server, it must revoke the signer on the spot, and write back “a proof of malicious work” ($\{\langle x, t, v, sig, S \rangle, \langle x, t, v', sig', S' \rangle\}$) to all servers.⁴
- Servers must keep all versions of values with timestamps.
- Servers can deny excessive write requests to mitigate a DoS type attack.
- The quorum system intentionally slows down the process of issuing new certificates.

Besides the above fundamental protocols, we have a couple of protocols to maintain the network configuration. Unlike above, these protocols are only between servers.

[join]

$s \rightarrow S :$ $Cert_s$
 $s_i :$ add s to QS_B
 $s \leftarrow S :$ return peer nodes of $s_i < Cert_{s_1}, Cert_{s_2}, ..., Cert_{s_n} >$

[leave]

$s \rightarrow S :$ $Cert_s$
 $s_i :$ remove s from QS_B
 // no response

Each node constructs a trust graph independently from the self certificate and ones returned by the “join” protocol. A node may issue the join request to nodes that were not included in the initial request. It repeats until collecting the certificates from all nodes in the graph. It is possible to fail to get the certificate from some nodes because of network failures or system problems, and in that case each node could see different graph. Even in that case, as long as the failure nodes are less than the quorum threshold (see the “Quorum System” below) the system will work properly.

Note that all transactions are signed by sender, and verified by receivers before processing the request. Therefore, no entities can issue join / leave for anyone else.

Transport Security

Transport security between clients and quorum members cannot be provided by ordinary SSL as the system does not rely on central authorities. We sign all messages between clients and each quorum member using certificates assigned to quorum servers and clients⁵. Secrecy is not necessary in theory - only integrity has to be guaranteed so that signed message (SM) byzantine quorum systems work, but encrypting messages could help mitigate some attacks in practice, so we encrypt and sign all messages by default.

⁴ This kind of “write back” might be blocked by attackers if they have enough power to control the Internet traffic but as the client keeps the revocation list, it will never accept any kind of message from revoked nodes.

⁵ PGP certificate in mind, which can have more than one signatures unlike X.509.

Quorum System

The quorum system consists of one or more cliques in the trust graph, each of which has at least $4b + 1$ nodes. As long as one of cliques is reachable from clients the quorum system satisfies the intersection property. Each clique has to be maximal and any two cliques do not share any node. We call cliques that satisfy those conditions quorum cliques (QC):

$$Q = \bigcup_i QC_i, QC_i \cap QC_j = \emptyset \text{ for } \forall i, j$$

A CQ can be b -masking quorum system and constructed with: $g(s, \bullet)$ with the start node s .

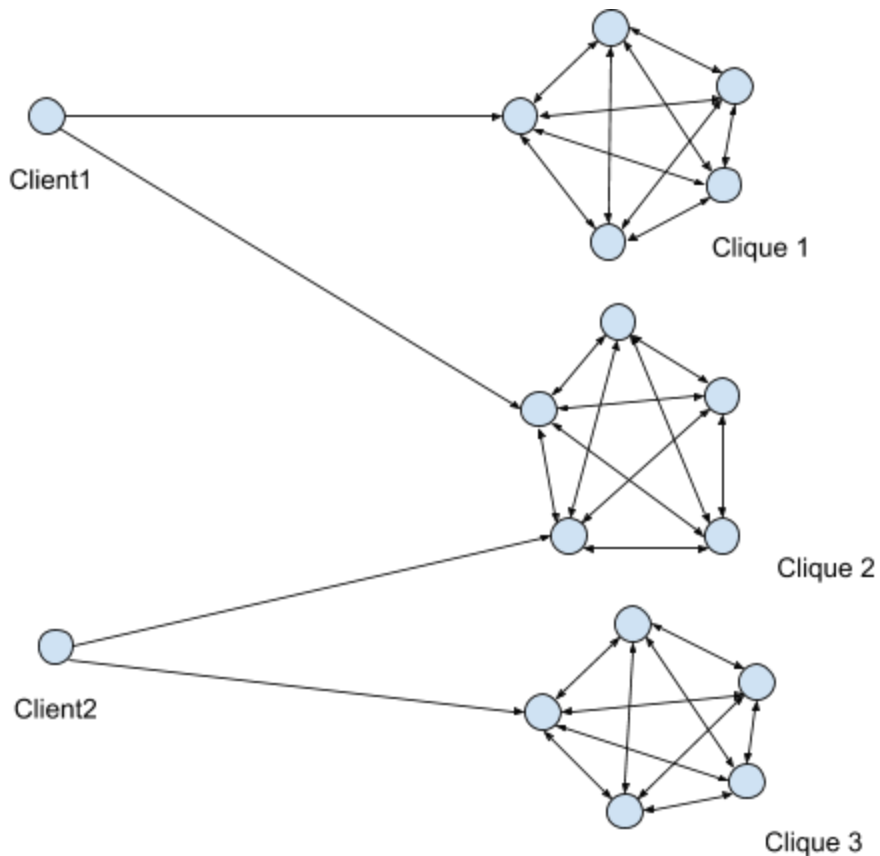
Algorithm

A function $g(s, L)$ that constructs all clique quorums that have distances less than L from s :

1. $V_s \leftarrow \{\}, queue \leftarrow \{(s, 0)\}$
2. repeat until $queue = \{\}$
 - a. $(v, d) \leftarrow dequeue$
 - b. if $d \geq L$ break
 - c. $QC_i \leftarrow findMaximalClique(v)$
 - d. for each edge e of v
 - i. $queue \leftarrow (e.v, d + 1)$ if it has not been visited
3. return $\{QC_i\}$

$findMaximalClique(s)$:

1. $C \leftarrow \{s\}$
2. for each $v \in V$
 - a. $C = C + \{v\}$ if both (c_i, v) and $(v, c_i) \in E$ for $\forall c_i \in C$
3. check that any $c_i \in C$ does not have a clique other than C
4. return C



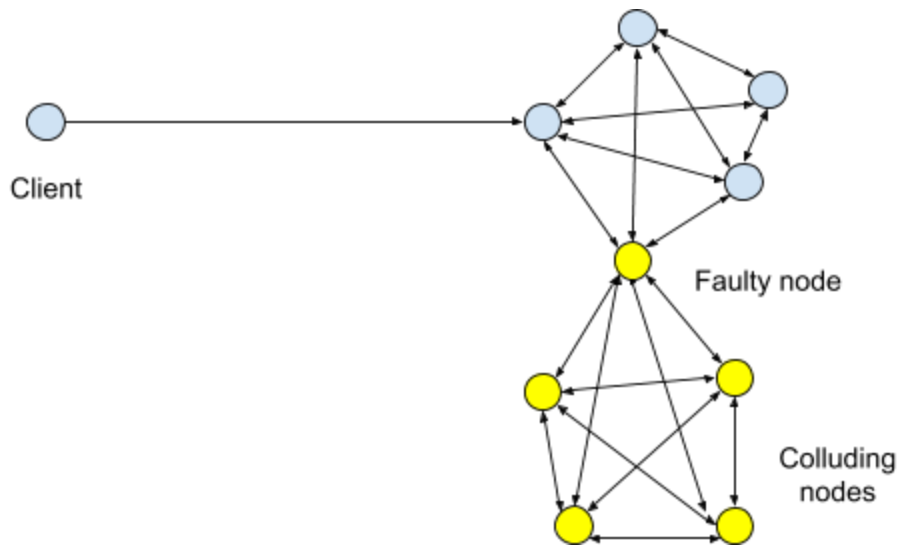
With the above algorithm, client1 has clique 1 and 2, and client 2 has clique 2 and 3.

Note that the graph is not representing network topology. Only assumption is that each node can be reached with the URL. In other words, failure nodes will not block any connections once the trust graph is established as servers do not talk each other.

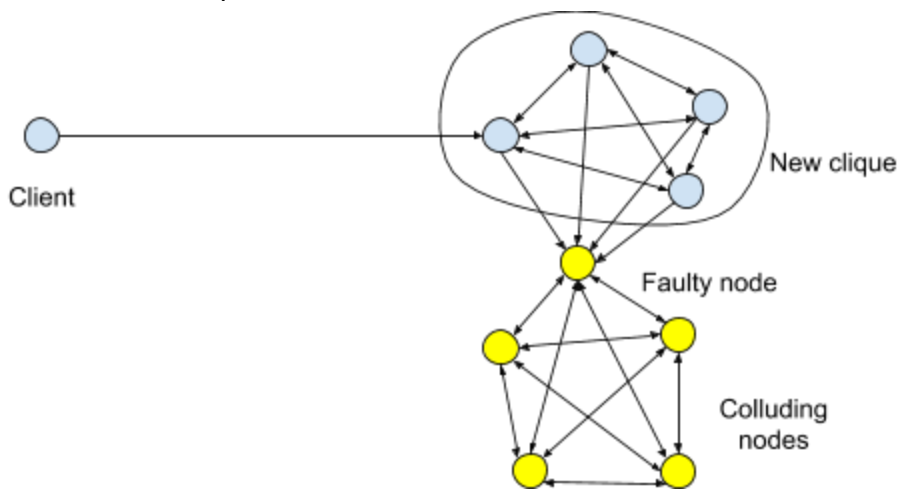
Any node can join the network without authentication, but it cannot join any existing clique unless it gets signed by all members of the clique. Put it in another way, unauthorized nodes can be members of the kv quorum, but they cannot join the authority quorum.

Sybil attack

As above, anyone can make up as many nodes as they want and have them join the network but those nodes cannot outnumber of authorities unless some amount of existing nodes in the cliques are compromised. Also we show the quorum system is immune to the sybil attack even in that case. Faulty nodes may try to outnumber the non-faulty nodes by making cliques:



But with the algorithm (g) any node cannot be a member of more than one clique, therefore the compromise node cannot extend the clique with faulty nodes. If it wants to make a faulty clique, the faulty node has no choice but severing links to all members itself, which results in kicking itself out of the clique.



Our quorum system requires that all quorum cliques need to agree on $\langle k, v \rangle$ to accept the value. Even if the client has a faulty clique it will not be deceived but it is possible to sabotage normal operations. We take on the safe side. The client can choose which cliques are trustworthy and exclude other cliques from the behavior of each clique. See below for revocation.

Revocation

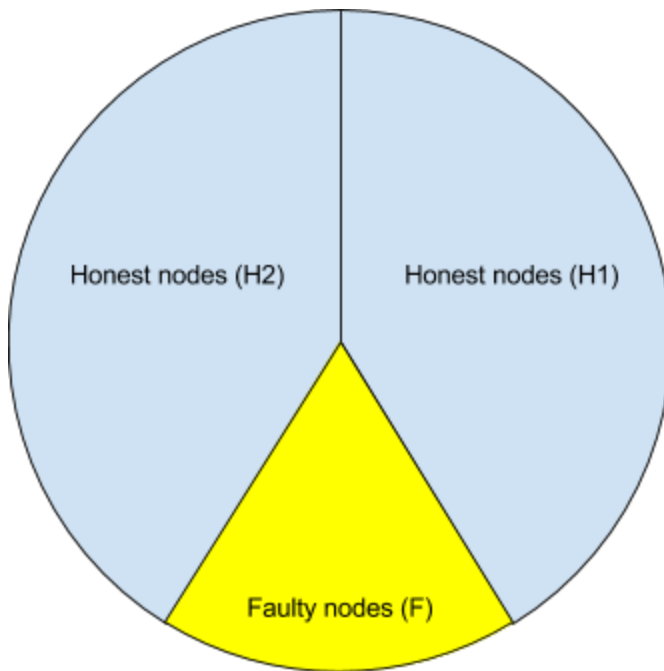
Revocation is the only way to keep the system sound in the long run. Even if the quorum system excludes uncertified keys by majority vote, excluding compromised nodes will increase the fault tolerance rate drastically.

Each node severs the trust link independently without consulting others. It is detected in both “read” and “write” protocols when a node signs both $\langle x, t, v \rangle$ and $\langle x, t, v' \rangle$ where $v \neq v'$. 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. See the security analysis for the detection rate.

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 attack.

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 colluding nodes, the attack will succeed.



The attacker sends echo requests to $\{H_1 \cup F\}$ and $\{H_2 \cup F\}$ separately:

$$\begin{aligned} c \rightarrow H_1 + F & : \text{“echo” } \langle x, t, v \rangle \\ c \rightarrow H_2 + F & : \text{“echo” } \langle x, t, v' \rangle \end{aligned}$$

The maximum number of signatures dishonest clients can get is $b + (n - b)/2$. Therefore, to overcome the equivocation attack we need $n - b > b + (n - b)/2 \Rightarrow n > 3b$.

Accepting signed data on write

Each node that has received the write request verifies S with cliques that cover itself. If S includes enough signatures signed by the members of the cliques, the node will accept the message: $ACCEPT(< x, t, v, sig, S >) = \forall QC_i \in g(self), |S' \cap QC_i| \geq b_i + 1$ where

$$S' = \{s \in S \mid Verify(s, < x, t, v, sig >) = true\}.$$

[Note: nodes in a KV quorum but not in an Authority quorum, i.e., $Q \in QS_B \setminus QS_A$, can make small cliques that have the size of less than 4. Those nodes will not be involved in signing therefore they are excluded from the above cliques.]

Equivocation check has to be done on write if a value for $< x, t >$ is already stored in the node. If the new request has the same variable (x) and timestamp (t), the node will not only reject the message but revoke all signers including the issuer who have signed on both $< x, t, v >$ and $< x, t, v' >$. It is unlikely to be able to detect malicious action here when malicious clients issue the write request -- they should carefully choose the sets of servers that will not intersect each other. However, the write request can be issued on the read request as writeback, and in this case it is possible that a set of nodes receives a write request $< x, t, v', sig', S' >$ which conflict with $< x, t, v, sig, S >$ stored in the storage. See below for the detection rate.

Detecting equivocation on read

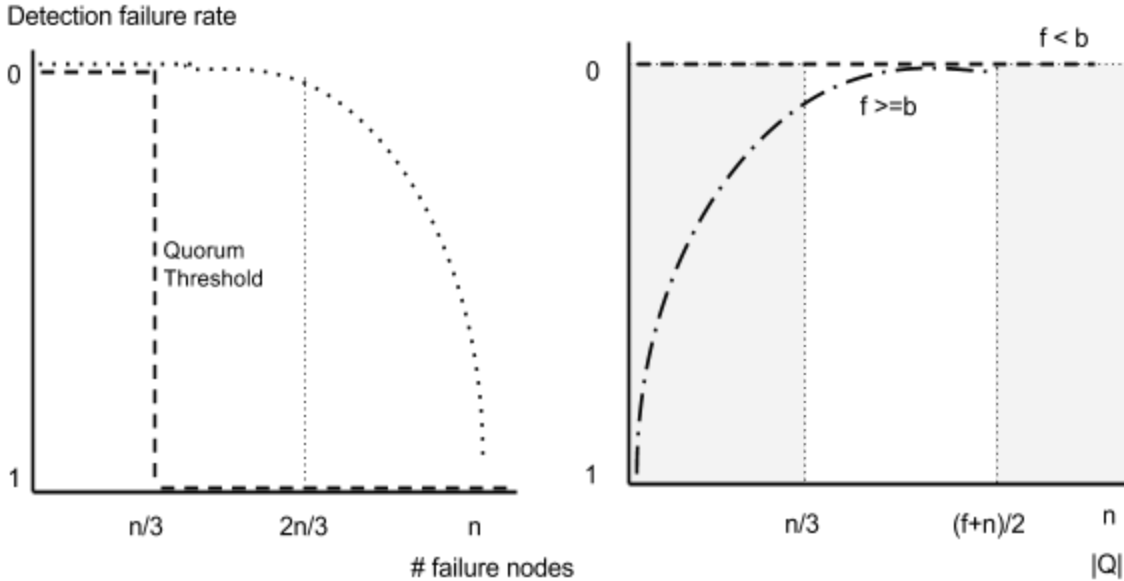
The system is intact as long as each clique keeps more than $2b + 1$ non-faulty nodes. Dishonest clients + colluding servers cannot do either 1) getting in the way for honest clients to read values, 2) deceiving honest clients with equivocation. When the number of faulty nodes exceeds the threshold, it is possible that honest clients no longer can collect enough number of valid responses.

From the threshold to n ($= |U|$) all servers including non-faulty servers will accept forged values, however, it is still possible to detect malicious actions on read by choosing a quorum randomly. The failure probability:

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

when $f > b$ and $|Q| < (f+n)/2$ where f is the number of faulty nodes,

assuming the size of H_i , $i = \{1, 2\}$ are the same.



In the case of $f < b$ 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., $f \geq b$, it will be possible that the client cannot detect the malicious action. 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.

After discarding those f nodes, if there are still more than $b+1$ valid $\langle x, t, v, sig, S \rangle$, the read operation will succeed and return the value. Nodes that are involved in the malicious action will be revoked immediately, which will change the quorum system and thresholds.

Applications

Decentralized PKI using a collective consensus protocol

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 Signal⁶. 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

⁶ <https://whispersystems.org/docs/specifications/x3dh/>

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.

Key lookup service

The system can be used to enhance security of the HKP⁷ service. HKP is a simple dictionary server; a user looks up PGP keys of recipients with email address, user ID or key ID. The sender decides if she trusts the PGP key by who signed the key. If there is no signer the sender trusts, it is up to her if she goes ahead. The system helps her decision.

A typical scenario: Service provider (SP) calculates a VRF index⁸ from the user ID to keep their privacy and publish the VRF public key to everyone. SP registers their PGP keys on behalf of users to the system: $WRITE(Q, vrf, H(PGP))$. When a sender looks up the key, SP will return the VRF index along with PGP. Users can verify the PGP by $READ(Q, vrf) = H(PGP)$. The sender also can double-check the log and signers if she wants to.

Email (or client apps) clients, on behalf of users, can check periodically that the value is not altered by someone else. Even if the $\langle k, v \rangle$ slot is protected by the quorum certificate, it will be a good practice for users to check it from time to time. She will know who did the unauthorized mutation from the signers of the key. Malicious actions retains in the log forever in all quorums.

⁷ <https://tools.ietf.org/html/draft-shaw-openpgp-hkp-00>

⁸ S. Goldberg, et. al., “Verifiable Random Functions (VRFs)”
[<https://tools.ietf.org/html/draft-goldbe-vrf-00#ref-SECG1>]

Transparency Framework

The system is also useful for a transparency framework such as CONIKS⁹. CONIKS uses Merkle tree to check if the sender and receiver see the same key set. Since the system guarantees $READ(Q_1, x) = READ(Q_2, x), \forall Q_1, Q_2 \in QS$ all trees constructed from the quorum system must be the same, i.e., any non-malicious CONIKS node will maintain the same tree which will serve well to keep its failure probability as low as expected.

Tree Head Store

Another example of a use of the system is to store the Merkle tree head in a specific slot so everyone can see the same value and log. This example can be more private than above. Only service provider maintains the user's ID and key, and construct Merkle tree. Only the tree head is published publicly. On the other hand, security relies on the entity that maintains user data.

E-mail address proof

To make the TOFU assumption work, service providers should verify the email address proof that shows the email address belongs to the user. The actual method is out of scope. This proof helps the system mitigate a DoS type attack as well.

Recovery from the key-loss situation

A slot accepts only data that is signed by the first writer. If the key of the one is lost, the slot will never be updated. We recommend to generate more than one keys and have them sign (trust) each other, and to store those keys in difference devices. This way, the slot can be updated with a signature by another key that was signed by the primary key. Also if you trust a key service provider, you can sign the key of the key server so the key server can update your slot on behalf of you when your key is lost.

Appendix

Implementation Notes

We define (go) interface for:

- Quorum system
- Transport layer (including transport security)
- Node (certificate)
- Crypto package

⁹ <https://coniks.cs.princeton.edu/about.html>

- signature scheme
- message encryption / signature
- collective signature
- Keyring
- RNG
- Storage

We implement (as of now):

- Quorum system with WoT
- Transport with HTTP
 - Transport security with PGP message encryption / signature
- Certificate with PGP key
 - Trust graph to manage nodes
- All crypto functions with PGP
 - PGP keyring to store certs, private key and revocation list
 - PGP key for certificate
 - PGP signatures in PGP key to construct the graph
 - PGP encryption / signature for transport security
 - PGP signature to sign $\langle x, t, v \rangle$
 - PGP signature for collective signature
 - PGP User ID for the URL
- Storage with (plain) Unix file to store the value with the filename: "variable (hex string).timestamp"

In golang, we use "golang.org/x/crypto/openpgp" for the PGP operations. Except PGP, we use the standard library only.

All messages are encrypted with the PGP key of recipients. With the PGP encryption scheme, a message is encrypted only once with all recipient's keys, then signed by the self key. The same PGP packet is sent out to each recipient.

The collective signature is just a series of the PGP signature packet in the current implementation. The reason why the interface defines the collective signature separated from the signature scheme is because of a possibility of replacing it with a threshold signature scheme in the future.

PGP Key

The following PGP packets are used in the system.

Public-Key Packet

Must include the primary public key. The key is used to verify the signature to make the trust graph within the quorum system.

Signature Packet

Represents trust incoming edges from signers. Must include the self-signed signature as specified in OpenPGP [ref].

Sub key packets

Inside the signature packet, at least one encryption key has to be included. The system uses the key for transport-security as well as encrypting messages.

User ID Packet

Must include a unique ID. For end users, the ID must be an email address. For servers, the ID can be an URL, UUID or PGP fingerprint.

User Attribute Packet

With subtype = 101,

Can include any data necessary for “email address proof”, e.g., DKIM, SAML, OpenID.

Revocation List Packet

A list of PIDs the node no longer trusts while it trusted before (therefore it signed the PGP certs).