Scalable Strongest-Link Decentralized Authorities

Iulia Tamas
Adviser: Bryan Ford

Abstract

Certificate, logging and time services play an important role in our online infrastructure. As these security-critical authorities rely on a few centralized servers, they are only as strong as their “weakest link”. We propose using collective authorities, or coauthorities, a different architecture that enables thousands of participants to collectively sign, or witness the public actions of an authority. The participating servers organized as a communication tree collectively sign and verify entries proposed by the root. We implemented and evaluated a prototype coauthority supporting logging and timestamping and showed that over 4000 participating servers distributed across the world can carry out collective signing with latencies of a few seconds.

I. INTRODUCTION

Applications and services from our online infrastructure rely on authorities for security-critical operations. Certificate authorities attest that a public key is owned by the subject named on a certificate [4]. Timestamp servers attest that a document was created or modified at a given time[1]. Directory authorities keep track of the servers that can take part in the execution of a specific application [6,14].

As these authorities oftentimes depend of a few centralized servers, they become targets for malicious adversaries. For example, attackers of CA authorities could steal their secret keys, impersonate various websites and monitor their users’ activity. Current solutions like Certificate Transparency [9,10], and splitting authority in a small consensus group [15,16] only offer retroactive defense or remain potentially vulnerable to state-level hacking [8,12], respectively.

With coauthorities we can move from “weakest-link” security to “strongest-link” security by splitting trust across many independently-run services. Each server in the coauthority contributes a share to a collective digital signature for each public output that it verifies. If a server identifies misbehaviour, it raises an alarm.

The main contribution of the paper is to show that coauthorities are scalable across thousands of servers in real-life situations. In order to limit computation and network bandwidth costs for the participating servers we multicast our protocol over a tree-based communication structure. We also accommodate a small number of server failures by including them in exception lists that are sent along with the protocol messages during collective signing, as to allow the participating servers to remove the failed servers share contributions from the collective key.

II. PERSONAL CONTRIBUTION

Being part of the Yale DeDis Research group, my contributions to Dissent were done as part of a group effort. I first worked on the Collective Signing Protocol (CoSi Protocol), then on implementing applications that used Collective Signing as to help evaluate the protocol, and then on issues
regarding achieving Collective Consensus. My work is a result of a direct collaboration with one other Yale undergraduate, and a sometimes direct, other times indirect collaboration with the DeDis Research group members, under the supervision of Professor Bryan Ford. Our work resulted in a paper that can currently found here:  

I. Work Items

1. Implemented Schnorr Signatures and the 4 phases of the Collective Signing protocol
2. Implemented Exception Lists and evaluated Collective Signing with Exception Lists on DeterLab
3. Implemented a layer over the Signing Nodes that would allow fine-grained, per round-phase, simulation of failures, to ensure the protocol implementation was robust
4. Implemented a TimeStamper application for local testing, and performed extensive local tests throughout the project
5. Collaboratively implemented a view change mechanism
6. Collaboratively implemented a voting protocol, and a group evolution application
7. Actively involved in decisions regarding the design of the networking level, and of the overall architecture

III. Signatures

Our Collective Signing Protocol is blind to the type of signature used. We chose Schnorr signatures as they are simple, and well understood [2,11,13].

I. Schnorr Signatures

To obtain a Schnorr Signature, first we chose a generator $G$ of a group of prime order $q$, in which solving the discrete logarithm problem is hard. A user then chooses a secret private key $x < q$, with the corresponding public key $X = G^x$. 

This (private,public) key pair matters in as much as it can be used to sign a message $M$, during a Signing Round, in a way that can be verified non-interactively [5,7]. To achieve this, the prover picks a secret $v$, then computes and sends a point commit $V = G^v$ to the verifier. The verifier challenges the prover with $c = H(V|M)$, and the prover computes a response $r = v - cx$. The pair $(c,x)$ is the Schnorr signature, and it can be verified by a third party using the signer’s (prover’s) public key $X$. If all is correct $U = G^r X^c$ should be equal to $G^{v-cx+xc}$ which would be equal to $V$. As such, it is enough for the third party to compute $H(U|M)$ and see that it is equal to the challenge $c$, to be convinced that the signature is correct.

II. Schnorr Multi-Signatures

The section above illustrates the general principal of how Schnorr Signatures work. What we used for our cothorities’ Collective Signatures is Schnorr Multi-Signatures. Now we can imagine our cothorities organized in a tree structure, with the branches signalling the possibility of a direct

---

1The paper can be found at: http://arxiv.org/pdf/1503.08768.pdf
network communication between two connected servers. Each coauthority has its own public key, and will generate its own secret for each signing round. As such our $X$ and $V$ are now replaced by $X_i$ and $V_i$ and represent the combined (multiplied) public keys of all the nodes in the subtree rooted at coauthority $i$. Starting at the leaves, the nodes compute the combined public keys and secrets and pass them up to their parents, until they reach the root which issues a challenge. The nodes then similarly compute and aggregate a response for that challenge in a bottom-up fashion. Each $r_i = \sum_{j=1}^{k} r_j$ where $r_j$ are the children of node $i$ in our tree representation. Each node has the ability to test that the pair $(c, r_i)$ verifies $X_i$ and $V_i$ for its own subtree, by performing an equivalent check to the one in the section above.

IV. Methods

I. Communication tree

We use a communication tree similar to those used in multicast protocols [3], in order to distribute the cost of computing multi-signature across a series of servers.

We implemented various of methods for organizing the servers in a tree structure, from manually generating static tree configurations, to dynamically generating graphs and choosing their minimum spanning trees as our communication trees. My contribution here was two-folded. First I created a Communication Tree given a list of Deter Lab machines, a desired number of hosts (nodes) per machine, and a desired depth. The additional constraints were that the tree is symmetrical and complete on all levels, except sometimes the last level, and that two adjacent nodes were not on the same machine. I achieved this using a simple 2-color greedy coloring algorithm. Second, by collaboratively implementing a view-change mechanism, I contributed code to the rehanging of a tree based on the new leader, as described in the ViewChange section.

We assume that once the communication tree is established it is known and accepted by all participating nodes. In addition to knowing the tree roster, each node also knows the public keys of all other nodes.

II. Tree Based Collective Signing

A round of the CoSi Protocol\footnote{The format of the messages used in the CoSi Protocol can be found in section viii} has 4 phases, executed during 2 round trips from the the leader, root of the communication tree, down to its bottom-most descendants, the leaves.

First The Announcement Phase. The leader multicasts down an AnnouncementMessage containing $M$, a message to be signed. The children first check the Announcement against application-specific requirements, and then propagate it downward to their children, recursively, until all nodes in the tree have received the announcement.

Second The Commitment Phase. Upon receiving an announcement message, a node enters the Commit Phase, which leads to a Commitment Message being sent upwards to a node’s parent. The commit phase consists of a node generating a round-lasting secret($v_i$), and a commit point based on that secret($V_i$). Before doing anything with these values, a node waits for its children’s Commitment Messages, containing $V_i$ and $\hat{V_i}$. $V_i$ is the commit point for this round generated by child i. $\hat{V_i} = \prod_{j \in \text{child}(i)} {V_j}$. After receiving all commitment messages from
its children, a node computes its own \( \hat{V} \) value, packs it with its \( V \) value in a Commitment Message, and send it upwards to its parent.

**Third** The Challenge Phase. After the root receives all Commitment Messages from its children, it multiplies them together to get \( \hat{V}_0 \). It then generates a challenge, equal to \( H(\hat{V}_0|M) \). The root node then packs this challenge in a Challenge Message and broadcasts it down to its children, as to reach all of its descendants, just as in the case of the Announcement Messages.

**Fourth** The Response Phase. Upon receiving a Challenge Message, a node enters the Response Phase, which leads to a Response Message being sent upwards to a node’s parent. The Response Phase consists of a node generating a response, \( r = v - xc \), as described in the Schnorr Signature section. Before doing anything with \( r \), a node waits for its children to send it their Response Messages containing \( r_i \) and \( \hat{r}_i \); \( r_i \) is the response for this round generated by child \( i \); \( \hat{r}_i \) is defined as \( r + \sum_{j:child(i)} r_j \). After receiving all response messages from its children, a node can compute its own \( \hat{r} \) value, pack it with its \( r \) value in a Response Message, and send it upwards to its parent.

After the 2 round trips are done, the root node has an aggregate response \( \hat{r}_0 \) with which it can verify the integrity of the signature by performing a check equivalent to the one described in the Schnorr Signature section. First compute \( U = g^{\hat{r}} * \hat{X}^c \). If \( H(U|M) \) is equal to the challenge \( c \), then the Schnorr Multi-Signature is complete, correct, and verified. Moreover, each of the non-root nodes verifies the correctness of their partial Schnorr Multi-Signatures as suggested in the Schnorr Multi-Signatures section.

### III. Exception Lists

We tolerate a small number of exceptions (\( O(\log N) \), where \( N \) is the number of participating servers) by adding Exception Lists in the protocol messages. The Exceptions can include any of the nodes except the leader, whose situations is discussed separately under View Changes.

The Exception List serves to identify by public key all the failed nodes. If a node fails before it managed to commit a signature share to the collective signature, we solely add its public key to the Exception List in its parent’s commit message, to take note of its failure, and ignore it for the rest of the round. If a node fails after it has already committed a share to the collective signature, we both add it to the Exception List and to a cumulative \( \text{ExceptionV\_hat} \) and \( \text{ExceptionX\_hat} \), which store the cumulative point commits and public keys of the excepted nodes, respectively. In this case, the Exception List, \( \text{ExceptionV\_hat} \) and \( \text{ExceptionX\_hat} \) are added to the parent’s response message.

When a node notices a child’s failure (eg. it timeouts on waiting for a response from that child), it adds to the Exception List the public keys of the child and of all the nodes in the child’s subtree. The Exception List that a node passes upward in the Commit and Response Messages, contains both the exception that the node itself notices, and the aggregated exceptions that the node’s descendants have noticed. Same holds for \( \text{ExceptionV\_hat} \) and \( \text{ExceptionX\_hat} \).

By allowing exceptions we affirm that we are willing to accept that a few node’s share to the collective signature for the current round be removed. As such, when verifying a signature produced in a round with exceptions, we first remove the \( \text{ExceptionX\_hat} \) for our subtree from the combined public key for our subtree. Then we remove the \( \text{ExceptionV\_hat} \) for our subtree from the combined public key for our subtree.

---

3. \( \hat{X} \) is the product of the public keys of all the nodes’ descendants, which a node can compute based on the group’s roster.

4. The structure of the Commit and Response Messages can be found in section viii.
point commits for our subtree. Then, we perform the same signature verification as described in the Tree Based Collective Signing section.

IV. Simulating Failures

In order to make sure our protocol is robust, and in order to be able to test that it works with exceptions (when nodes occasionally fail at random points throughout the protocol), I implemented an overlaying structure over our nodes that would allow fine-grained, controlled, failures during specific phases. We had two ways of testing with failures. The first way ensured the failures were deterministic. Our nodes were embedded in FaultyHosts, which had as an additional field a map that would specify a per round failure type which could be “commit,” “response” or “null”; the failure type would indicate whether a node is required to fail on commits, responses or not at all. The second way of testing allowed the failures to be non-deterministic, and offered the nodes more freedom in when they would fail. We embedded in nodes failure-probabilities and failure-rates. When using failure-probabilities a node would fail each round with a certain probability. When we used failure-rates, a node would fail every $n^{th}$ round, when its failure-rate equalled $n$.

Failures were simulated as described above, and tested on Deter Lab. A potential failure of the root node, however, was not considered here, as a failed root would require a reconfiguration of the communication tree. We consider this case separately in the View Change section.

V. Signing Modes

To show the versatility of the signing nodes, and illustrate how they may be used in real-life applications I implemented three signing modes.

V.1 Simple Collective Signature Mode

When running in this Simple Collective Signature Mode, the root (leader) node would start a Signing Round whenever it required a message to be signed. The non-leader nodes would follow through with the protocol without any additional contributions to the message.

V.2 Merkle Key Mode

When running in Merkle Key Mode, each node can contribute to the message signed for the round. The high level picture is that a round message is now seen as a log which the leaves create, hash, and send to their parents as their local Merkle Tree Roots. The parents, in turn, add the information they want to commit to the children Merkle Tree Roots, hash again, and create their own local Merkle Tree. Then, they send their Merkle Tree Root up in a Commitment Message. The process continues until the root has the overall Merkle Tree Root.

The specifics of how the Merkle Tree Roots are created, and what they can represent can be found in the TimeStamper Application and Clients section.

V.3 Collective and Private Signature Mode

This mode is similar to the Simple Collective Signature Mode. What was added to the simple mode, is allowing a node to digitally sign its log before sending it up in a commitment message.

We needed the additional digital signature in the voting protocol, where we wanted the identity of a vote’s caster, and the time the vote was cast to be strictly and certifiably interrelated.
VI. TimeStamper Application and Clients

I implemented TimeStampers, which were servers with Signers embedded (Signer is the name of the interface which a server able to participate in CoSi implements). I also implemented Clients, which contacted TimeStampers of their choice and sent them hashes of messages they wanted timestamped and signed. The TimeStampers collected messages from their Clients and every round_time (generally a few seconds), the leader time-stamper launched a signing round. Upon receiving an Announcement for the new round, a timestamper would call an AggregateCommits function. This function took the array of messages the timestamper had collected from clients, considered them to be the leaves of a new merkle tree, and computed the entire local-merkle-tree, bottom-up, as well as the associated local-merkle-proofs for each leaf.

This local-merkle-root represents the node’s clients’ aggregated contributions for the Log of the current signing round. This contribution had to, however, be integrated with all the MerkleRoots the children nodes had produced for their rounds. As such, a node would again make use of merkle trees. A node would create an array of all its children’s merkle roots, its own merkle root, and a hash of the current round information (point commits, exceptions, etc...). The entries in this array were considered the leaves of a new Merkle Tree the node created.

The new Merkle Root was then embedded and sent up in a Commit Message. The need of merkle trees and paths, however, is not complete until the node receives the Challenge Message for the round. The Challenge Message a node receives contains a Merkle Proof that shows that the node’s commitment (its merkle root) was integrated in the overall merkle root, and signed on, collectively.

Now, the node has two things to do. First, for each child, the node takes the Merkle Proof it received from above, appends to it the Merkle Proof that leads from its own Merkle Root to the child’s Merkle Root, and sends it down to the child in a personalized Challenge Message. Second, for each client message, the node takes the Merkle Proof it received from above, appends to it the local-proof for the client-message and sends it in a personalized message to the corresponding client.

A different member of the team, starting from this implementation, created a TimeStamper implementation that fit the requirements and infrastructure we had set for testing on DeterLab.

VII. View Changes

We have described in previous sections how we handle Failures via Exception Lists. Those sections, however, did not account for a more special type of failure: leader failure.

When the leader fails, the communication tree must change to account for this failure, and gain another root. We accomplished this by implementing a heartbeat mechanism and a view change mechanism.

A new view consisted in the communication tree being rooted at a different node. As such, all connections between nodes stayed the same, but some node roles reversed from parent to child and the other way around. The new root was chosen by going round-robin through a sorted list of all participating nodes.

The views were identified by view-numbers, starting at 1. Each node kept track of the view it was on, and potentially, of the view it was trying to change to by increasing his expected-view-change number each time it entered the view-change mode.

The heartbeat mechanism, ensured that each node kept track of the last time it received a message from the root node. If this time since was greater than a preset timeout, the node entered in view-change-mode. The View Change mechanism, however, could only be initiated by the node
that identified itself as the root for the next view. This node would eventually also timeout on
waiting for a message from the current root, enter view-change mode and initiate the view change
mechanism.

While in view-change mode, a node would only accept messages related to view changes. A node
can also timeout while being in view-change mode, case in which it reenters view-change mode with
an expected-view-change-number increased by 1.

The View Change Mechanism consisted in devoting a round (or more if a quorum was not
established) to reaching consensus on the next root. The node initiating the view change multicasted
to its immediate peers (parent and children) a View Change message. Upon receiving this message,
the parents and children would propagate it forward, to ensure all nodes could become aware of
the proposed view change. If the nodes agreed with the view change, they would multicast back
View Accept messages. The next root waited for View Accept messages, and if it received a quorum
of such messages (more than 2/3 of number of nodes accepting the view change), it assumed its
new role as a root, and it multicasted a View Confirmed message to its peers. The View Confirmed
message was propagated through the tree, and upon receiving it, a node would switch to operating
on the new view (where likely its parent and children configuration was different).

VIII. Voting Protocol and Group Evolution

We implemented a voting protocol through which the nodes could cast votes and achieve consensus
on, for example, an action they wanted to collectively take. Our use case was in the implementation
of an application similar to a Directory Service, which required group evolution, and monitoring.
As such, nodes were able to use votes to decide on adding or removing a node from their roster of
nodes[6].

Vote Requests for removing nodes are always initiated by the leader, and embedded in an
Announcement Message. The protocol proceeds mostly the same as it normally would, with the
nodes waiting for Commit Messages form descendants before sending their commit message up.
The difference now is in the way a node aggregates the information in the Commit Messages it
Receives. All the VoteResponses, but the node’s own are appended within a single CountedVotes
struct. The node then counts the number of votes for and of votes against, adding its own vote to
the count, and fills in the corresponding 2 fields in the CountedVotes struct. Then, the node signs
the CountedVotes struct, and fills in the Signature field of its own vote response with the resulting
signature. This VoteResponse of node A, which A can now send up, allows any node B to verify
that A’s vote was added to the counted votes by A, immediately after receiving all votes from A’s
children. When the fully filled in CountedVotes comes down to the nodes, via Challenge Messages,
node A can also check whether its ancestors counted her vote for or against correctly, and that they
did not tamper with any of A’s children or descendant’s votes.

Vote Requests to add nodes are currently being implemented. They are initiated by the node
wanting to join the protocol, and sent up to the root. The Root then initiates a voting round, just
like it would when wanting to remove a node.

V. Evaluation

We evaluated on 32 physical DeterLab machines configured in a star-shaped virtual topology. On
each machine we ran up to 128 separate CoSi servers we will call hosts. A round-trip latency between	wo hosts on different machines was 100 milliseconds, as to simulate distances between cothority
servers that could potentially be distributed around the world. To enforce this we also disallowed

[6]The message structures used for Voting can be found in section viii
parents and a children servers being on the same machine, as explained in the Communication Tree section. Our code, written in Go, consists of around 7600 lines of server code and can be found at https://github.com/DeDiS

A different member of the team was responsible for carrying out the evaluation on Deter Lab. We looked at computational costs, latency, and scalability.

I will only include two graphs from our paper: Decentralizing Authorities into Scalable Strongest-Link Cothorities[7]. The first figure shows some of our results for scalability: latency increases gradually with the number of hosts, averaging at around 1.5 seconds for 4096 hosts. The second, displays the view change mechanism, the spikes corresponding to view changes. More of our graphs can be found in the paper referenced above.

VI. CHALLENGES

Keeping track of all the moving parts (signers, timestampers and clients) was fairly challenging. I had to be very careful with issues of concurrency, and make sure I had extensive modular tests to quickly pin-point from which part of the project the problem/bug was originating.

Figuring out what network level architecture best fit our dynamic ever-changing needs as our project grew more and more complex from a simple Collective Signing Protocol to a versatile protocol allowing use in three signing modes, by 3 types of applications, with voting, view changes, and heartbeats enabled was quite a challenge. Although I was not primarily responsible for implementing the network layer, in implementing the signing protocol I would unveil various situations, needs, and bugs that would lead to rethinking part of the networking layer. For example, introducing Timeouts, allowing exceptions, and simulating failures led to a change in design from a synchronous, parent waiting for all kinds of responses from all children, to a dispatcher getting the messages and feeding them on commit channels, response channels, and view-change channels; the signing nodes would listen on these channels for variable amounts of time, depending on the timeout enforced for a specific kind of message.

Working on my senior project was my first time using Go for a project that had more than 100-200 lines. It proved to be a very easy to use, friendly language.

VII. CONCLUSION

We have demonstrated a way of building strongest-link collective authorities. We managed to distribute trust across thousands of servers, while maintaining low latencies and low computational costs. This paper then serves as proof that we can get better security from our authorities.

VIII. CODE SNIPPETS

I. Signing Messages

type AnnouncementMessage struct {
    Messg []byte
    VoteRequest *VoteRequest
    Round int
}

type CommitmentMessage struct {
    V abstract.Point // commitment Point
    V_hat abstract.Point // product of subtree commitment points
    X_hat abstract.Point // product of subtree public keys
    MTRoot hashid.HashId // root of Merkle (sub)Tree
    ExceptionList []abstract.Point
    CountedVotes *CountedVotes // subtree’s votes
    Round int
}

type ChallengeMessage struct {
    C abstract.Secret // challenge
MTRoot hashid.HashId // the very root of the big Merkle Tree
Proof  proof.Proof   // Merkle Path of Proofs from root to us
CountedVotes *CountedVotes // subtree’s votes
Round   int
}

type ResponseMessage struct {
    R_hat abstract.Secret  // response
    ExceptionList []abstract.Point  // list of public keys
    ExceptionV_hat abstract.Point  // cumulative point commits
    ExceptionX_hat abstract.Point  // cumulative public keys
    Round   int
}

II. Voting Messages

type VoteRequest struct {
    Name   string  // name of server action is requested on
    Action string  // "add" or "remove"
}

type VoteResponse struct {
    Name   string  // name of the responder
    Accepted bool
    Sig     BasicSig  // Own Digital Signature on Own and Subtree Votes
}

type CountedVotes struct {
    Votes []*VoteResponse  // vote responses from descendants
    For   int  // number of votes for
    Against int  // number of votes against
}

IX. References