


A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light greenish-blue. They are set against a dark navy blue background with subtle diagonal lines.

Addressing Synchronization Problem in CTng Local Testing Environment



CTng – improved version of Certificate Transparency

- A redesign of traditional certificate transparency proposed by Prof Amir Herzberg

Goal : Achieve Certificate transparency and Revocation Transparency with the following Principles:

- 1) Not Trust Third Party: All the entities are not trusted in our network (Loggers, Certificate Authorities, Monitors and Gossipers)
- 2) Allow offline Validation
- 3) Denial of Service attack (DOS) resiliency



Digital Signature

- Cryptographic Output used to verify the authenticity of data
- Protection against tempering
- Sign the message using the secret key
- Verify the message using the public key
(with no knowledge of the secret key)

Certificate

A Certificate is issued by a Certificate Authority to the domain owner who have requested it and have proven its identity to the Certificate Authority

A simple example of a certificate for domain owner A:

("PK_A||A||Expiration Date", Signature(SK_CA, m))

PK_A: The Public Key of A

SK_CA: Secret Key of the Certificate Authority

m = "PK_A||A||Expiration Date"

Signature(SK_CA, m) = CA's signature over m using its Secret Key





With the certificate

Client Bob wants to establish a connection with domain A

A will send the certificate to Bob

Bob will then be able to verify the Public Key of A Using the public key of the CA

Since Bob knows the certificate ("PK_A||A||Expiration Date", Signature(SK_CA, m))

And the public key of the CA

Problem:

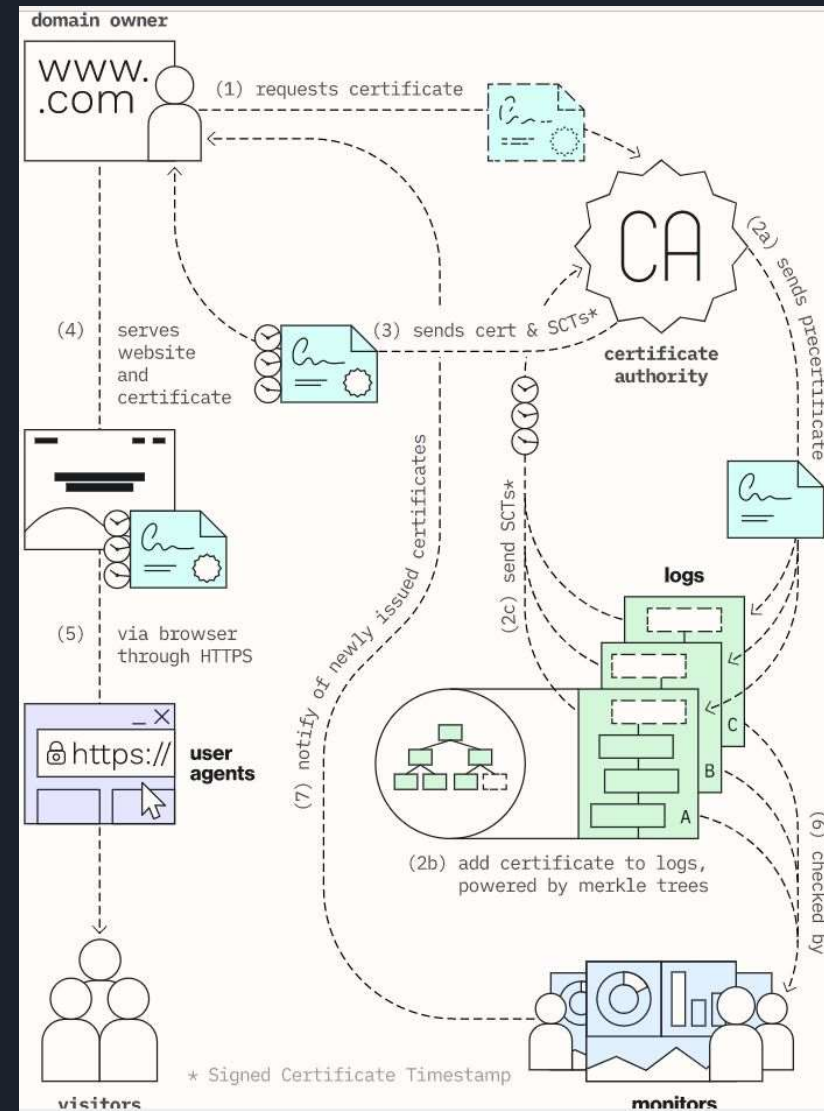
- 1) CAs are the Trusted Entities here
- 2) But they are essentially just businesses

Certificate Transparency

Each Valid Certificate will be logged by a logger using an append only structure (the Merkle Tree)

This allows public monitoring and auditing

- 1) Website owner request certificate from CA
- 2) CA issues pre-certificate (Certificate without CT info) and send it to loggers
- 3) Loggers return SCTs to the CA to show the commitment to add the precertificate within the MMD
- 4) CA attaches the SCTs to the Certificate as part of the message that will be signed by the CA and send to the domain owner
- 5) Domain owner then send the certificate to the browser



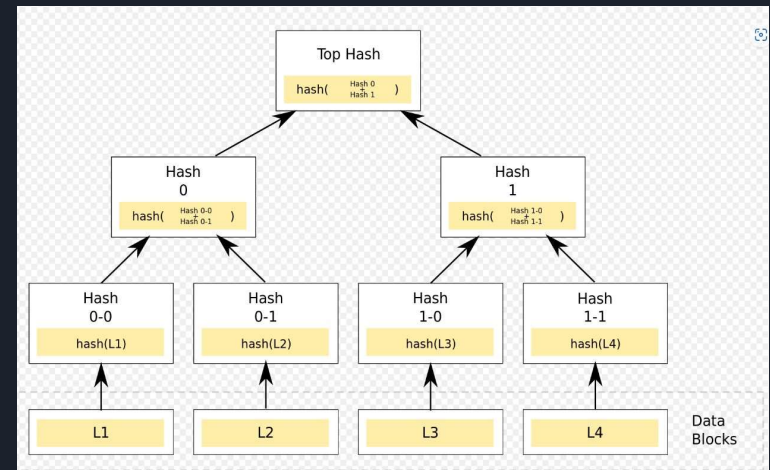
Monitor and Logger in CT

Logger (CT log)

- Anyone can query whether a certificate is included in a logger
- When the logger signs the Root Merkle tree (Periodically), it creates a Signed Tree Head (STH)
- $\text{Verify}(\text{Cert} + \text{POI}(\text{cert})) = \text{STH}$

Monitor

- Monitors can be ran by anyone
- A CT monitor is often a subscription service





CTng

Monitor → Monitor-Gossiper network with Threshold Signature

Threshold Signature Scheme (TSS)

A signer group where any subsets of (Threshold size) of the group can produce signatures on behalf of the group.

Multi Signatures

are threshold signatures that they reveal the identities of the group members who produced them. In multi signatures, the signing members are not anonymous at all.

DOS resiliency → TSS does not require all monitor's signature to generate a full signature

NTTP → TSS requires at least Threshold number of signature for full authentication

Offline Validation → monitor synchronization with group authenticated data for client

→ Client only need to query one monitor once per period to get all the updated info



TSS-BLS

Gen:

Sign:

Given a message m and the provided

secret key share k ;

each participant i generates a partial signature $S = H(m) \cdot k$;

Verify:

- 1) Verify partial sig (participants are exposed)
- 2) Verify Aggregated sig

```
[
  {
    "application": "CTng",
    "period": "0",
    "type": "http://ctng.uconn.edu/304",
    "signer": "",
    "signers": {
      "0": "localhost:8082",
      "1": "localhost:8080"
    },
    "signature": [
      "{\\\"sign\\\":\\\"921791199f1d019b19e12a1da2cd314cd0ebdbb455e8e83e0312768d7cc26c591e3c597050961ca3b473c2ed822fe910\\\", \\\"ids\\\":[\\\"localhost:8080\\\",\\\"localhost:8082\\\"]}\\\", \\\"\\\""}
    ],
    "timestamp": "2022-11-22T03:34:02Z",
    "crypto_scheme": "BLS",
    "payload": [
      "localhost:9000",
      "localhost:9000{\\\"Timestamp\\\":\\\"34\\\",\\\"RootHash\\\":\\\"e4fde9a570548f076a99e1d66e88ad80379a61970f9320f1880b922c17742829\\\",\\\"TreeSize\\\":0}\\\", \\\"\\\""}
    ]
  }
]
```

Overall structure (tentative)

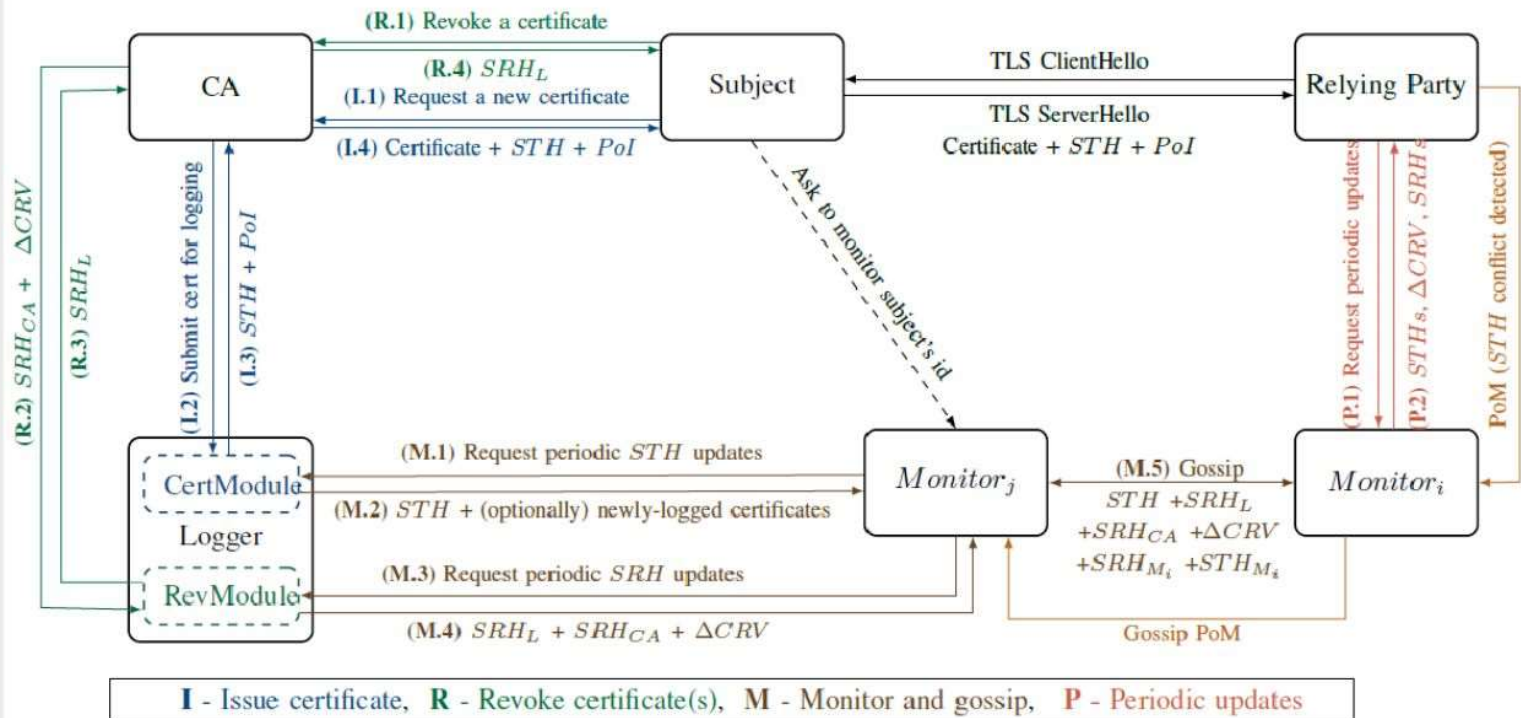
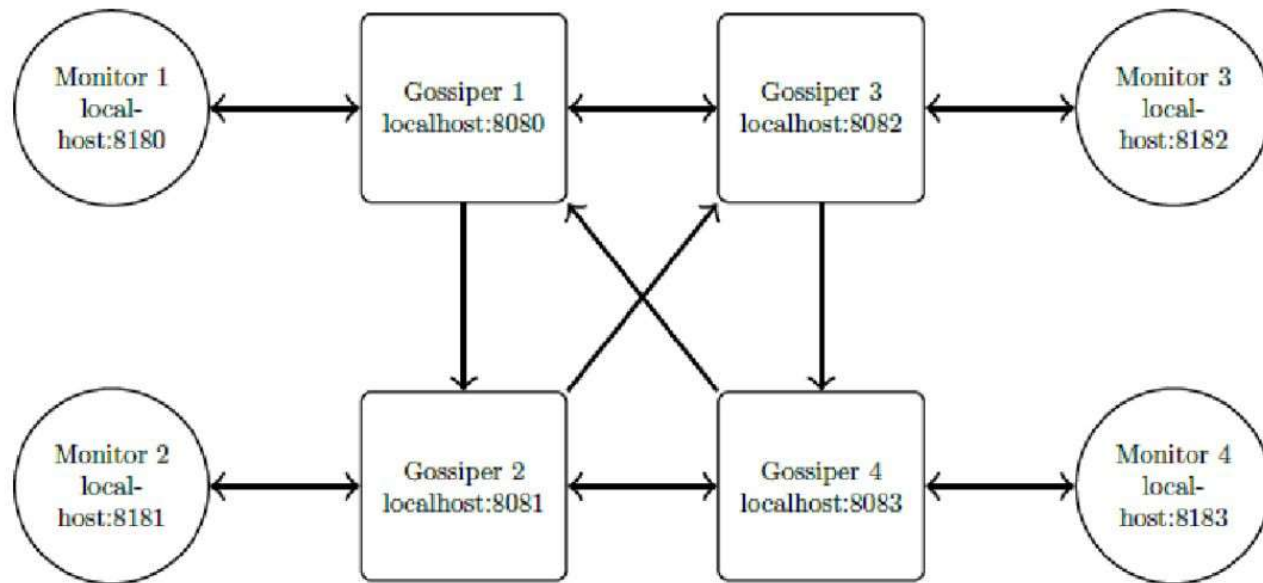


Fig. 1: CTng: entities and interactions. See §IV-A for an overview and Table II for notation.

This Project



Local Testing Network

Loggers and CAs

We are running a system of 3 CAs, 3 Loggers, 4 monitors and 4 gossipers

The port number for loggers and CAs are listed below:

Logger 1	9000
Logger 2	9001
Logger 3	9002
CA 1	9100
CA 2	9101
CA 3	9102

Monitor Network (Pull Based)

Monitor ID	Connected Loggers	Connected CAs
Monitor 1	Logger 1, Logger 2	CA1, CA3
Monitor 2	Logger 2, Logger 3	CA2, CA3
Monitor 3	Logger 1, Logger 2	CA1, CA3
Monitor 4	Logger 2, Logger 3	CA2, CA3



Note: In this Project

- 1) The monitors query the CAs for revocation information instead of the revocation module of the logger (because the design for both the CA and the Logger has not been finalized and has not been fully implemented)
- 2) Monitors only query monitors and CAs once period (Pull based)
- 3) We use different port on the machine to simulate different physical machines (each entity should be ran on a separate machine in real life application)



Goal of Monitor-Gossiper Synchronization

All the monitors must have the CTng information to serve the client, regardless of the CAs/loggers this single monitor queries.

E.g. Monitor 1 queries Logger 1

Monitor 2 queries Logger 2

Monitor 1 and Monitor 2 need to have the same group authenticated information from both Logger 1 and Logger 2 to serve the client

Monitor Query

Use sleep to make sure every Monitor Queries at the same time

```
func Getwaitingtime() int{
    timerfc := time.Now().UTC().Format(time.RFC3339)
    Seconds, err := strconv.Atoi(timerfc[17:19])
    if err != nil {
    }
    Seconds = 60-Seconds
    return Seconds
}
```

```
time_wait := gossip.Getwaitingtime();
time.Sleep(time.Duration(time_wait)*time.Second);
```

Periodic query is implemented by
time.Afterfunc

Syntax:

```
func AfterFunc(d Duration, f func()) *Timer
```

Here, *Timer is a pointer to the Timer.

Return Value: It returns a *Timer* which is then used to cancel the call with the help of its Stop() method.



File I/O

Uses the Golang os package

- provides a platform-independent interface to operating system functionality
- Unix-like design
- Go-like error handling (more information available)

The os interface is intended to be uniform across all operating systems. Features not generally available appear in the system-specific package syscall.

Use cases are in util/IO

Interface:

```
func ReadByte(filename string) ([]byte, error) {}  
func WriteData(filename string, data interface{}) error {}  
func CreateFile(path string) {}
```




“Multithreading” and Shared resources

Gossiper communication are push based

- Everytime the GossipData function is called, a new goroutine (light weighted thread) will be created
- Everytime a Gossiper receives an data object, the Handler will run through a duplication check before assigning the data received to the specific handlers
- Data-specific handlers will store the object before processing it for future duplication checks

Duplication check will need read access to the storage

Store object will need write access to the storage



Goroutine vs Thread

Goroutine	Thread
Goroutines are managed by the go runtime.	Operating system threads are managed by kernal.
Goroutine are not hardware dependent.	Threads are hardware dependent.
Goroutines have easy communication medium known as channel.	Thread does not have easy communication medium.
Due to the presence of channel one goroutine can communicate with other goroutine with low latency.	Due to lack of easy communication medium inter-threads communicate takes place with high latency.
Goroutine does not have ID because go does not have Thread Local Storage.	Threads have their own unique ID because they have Thread Local Storage.
Goroutines are cheaper than threads.	The cost of threads are higher than goroutine.
They are cooperatively scheduled.	They are preemptively scheduled.
They have faster startup time than threads.	They have slow startup time than goroutines.
Goroutine has growable segmented stacks.	Threads does not have growable segmented stacks.

Data Structure Transferred

Gossip Object

Application	String
Type	String
Period	String
Signer	String
Signers	Map[int]String
Timestamp	UTC-RFC 3339 String
Signature	[2]String
Crypto_Scheme	String
Payload	[3]String

Note:

- 1) we always allocate a string array of size 2 for signature(s) and payload(s), if only one is needed, we will set the second item in the array to be an empty string
- 2) Signers is an optional field, it is used when the Threshold Signature Scheme needs to know the signers for signature verification

Data being transferred in this project

```
const (  
  STH      = "http://ctng.uconn.edu/101"  
  REV      = "http://ctng.uconn.edu/102"  
  ACC      = "http://ctng.uconn.edu/103"  
  CON      = "http://ctng.uconn.edu/104"  
  STH_FRAG = "http://ctng.uconn.edu/201"  
  REV_FRAG = "http://ctng.uconn.edu/202"  
  ACC_FRAG = "http://ctng.uconn.edu/203"  
  CON_FRAG = "http://ctng.uconn.edu/204"  
  STH_FULL = "http://ctng.uconn.edu/301"  
  REV_FULL = "http://ctng.uconn.edu/302"  
  ACC_FULL = "http://ctng.uconn.edu/303"  
  CON_FULL = "http://ctng.uconn.edu/304"  
)
```

- 1) STH, REV are signed by Loggers and CAs
- 2) ACC is signed by the monitor
- 3) CON is not signed, but the payload of the CON contains 2 signature from the conflicting objects (So it can still be verified)
- 4) [wildcard]_FRAG are signed by the monitors in old design, by the gossipers in the new design
- 5) [wildcard]_FULL are generated by the gossipers by aggregating threshold hold number of partial signatures

Note:

- 1) CON_FRAG, CON_FULL, ACC Do not exist in the old design because the monitor will produce ACC_FRAG straight away and CON_FRAG, CON_FULL will never be generated due to fact that monitors are responsible for threshold signing while CON is generated by the gossipers
- 2) All data transferred are protected by digital signature and are therefore verifiable



Old Design - Monitor Each Period

- 1) Monitors query loggers/CAs (with no PoM held against) for STH/REV information
- 2) Monitors send STH/REV (or ACC_FRAG if Queried Loggers/CAs are not responding) to their connecting gossipers and wait for gossip_wait_time(A configurable constant that should be changed according to the information propagation delay in the network)
 - Generate STH_FRAG/REV_FRAG (= sign the STH/REV with Monitor's BLS partial signature) Otherwise and send to its gossipers
 - Do nothing if there is a PoM against the original signer of STH/REV
- 3) Receives STH/REV/CON(If applicable)/ACC_FULL(if Applicable) from its gossipers
- 4) Prepare the Client update, store all the group authenticated data (File I/O, json encoding)



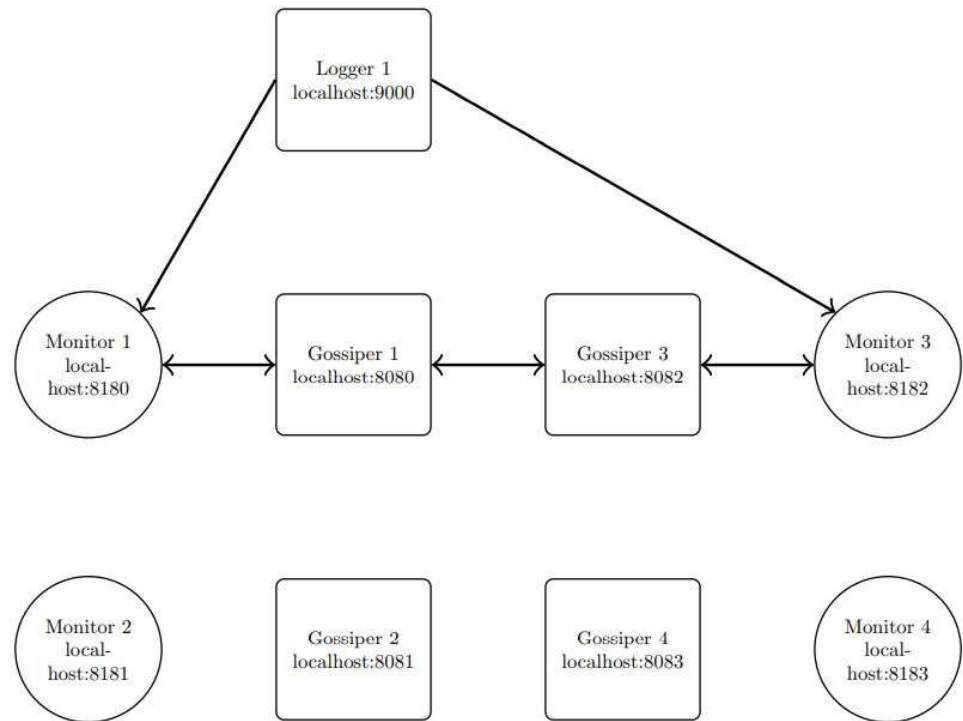
Old Design - Gossiper Each Period

- 1) Upon receiving any non-duplicate data, the Gossiper should first store it, invoke the data-specific handler and then gossip it to other gossipers it connects to
- 2) For any non-duplicate STH/REV/CON received (with Logger/CA's signature, not yet signed by any gossipers), the gossipers will send it to the monitor and gossip it its connected gossipers
- 3) For any non-duplicate STH_FRAG/REV_FRAG/ACC_FRAG received, the gossipers will increment its counter by 1 if the counter reaches the threshold while no fully signed version is yet stored, the gossipers should generate corresponding STH_FULL/REV_FULL/ACC_FRAG c and send it to the monitor (Also gossip it to other gossipers)

Old Version: Monitor responsible for Threshold Signing

Case Config

- Threshold = 2
 - Total monitor-gossiper = 4
- 1) If Logger 1 is an honest logger, it will send the same STH to both monitor 1 and monitor 3
 - 2) M1 can get STH from L1 and G1
 - 3) G1 can get STH from M1 and G3
 - 4) G1 can get STH_FRAG from M1 and G3





Problem:

Shared Resource on both monitors and gossipers

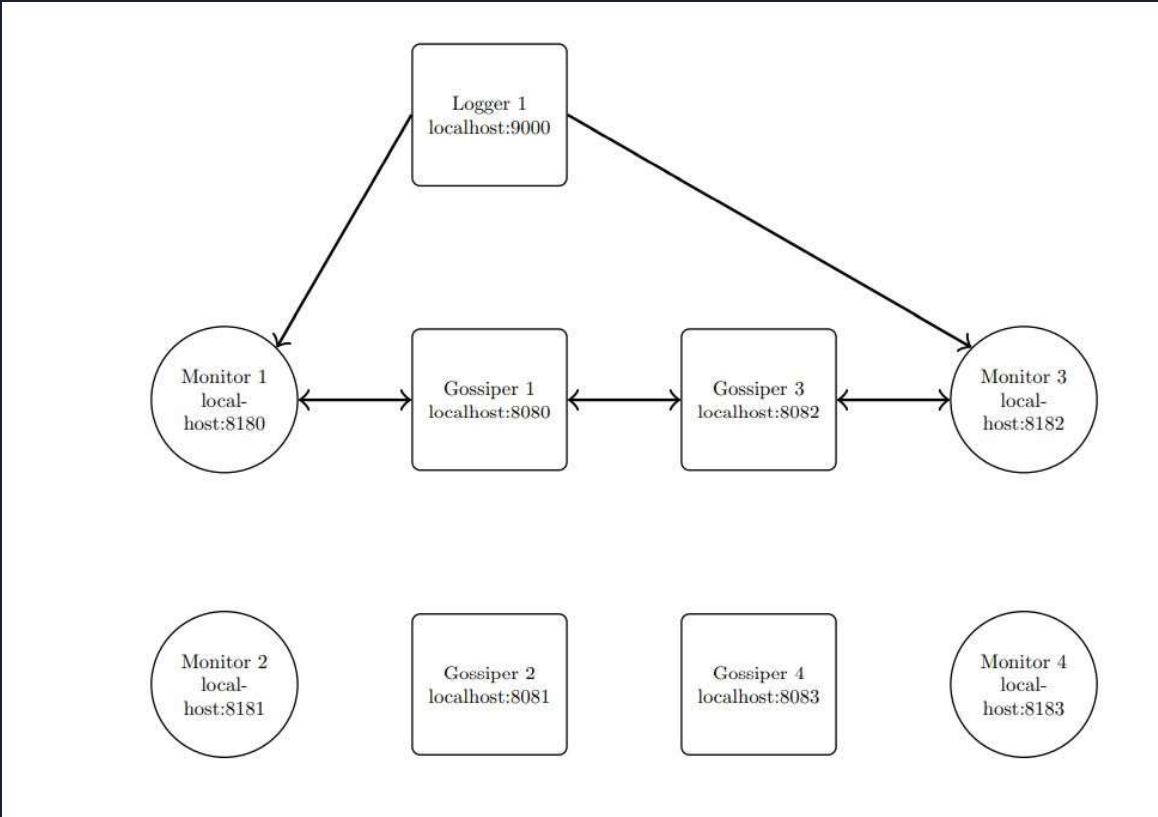
Duplication check will need read access to the storage

Store object will need write access to the storage

Many Goroutine will need read and write access to the storage (monitor and gossip)

The data structure we use for storage is map

Which is not a thread-safe implementation



Old design General Object Handler

```
func handleGossip(c *GossiperContext, w http.ResponseWriter, r *http.Request) {
    // Parse sent object.
    // Converts JSON passed in the body of a POST to a Gossip_object.
    var gossip_obj Gossip_object
    err := json.NewDecoder(r.Body).Decode(&gossip_obj)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
    // Verify the object is valid, if invalid we just ignore it
    err = gossip_obj.Verify(c.Config.Crypto)
    if err != nil {
        //fmt.Println("Received invalid object "+TypeString(gossip_obj.Type)+" from " + util.GetSenderURL(r) + ".")
        fmt.Println(util.RED,"Received invalid object "+TypeString(gossip_obj.Type)+ " signed by " + EntityString(gossip_obj.Signer) + ".",util.RESET)
        http.Error(w, err.Error(), http.StatusOK)
        return
    }
    // Check for duplicate object.
    stored_obj, found := c.GetObject(gossip_obj.GetID())
    if found && gossip_obj.Signer == stored_obj.Signer{
        // If the object is already stored, still return OK.
        //fmt.Println("Duplicate:", gossip_obj.Type, util.GetSenderURL(r)+".")
        //fmt.Println("Duplicate: ", TypeString(gossip_obj.Type), " signed by ",gossip_obj.Signer+".")
        err := ProcessDuplicateObject(c, gossip_obj, stored_obj)
        if err != nil {
            http.Error(w, err.Error(), http.StatusOK)
            return
        }
        http.Error(w, "Received Duplicate Object."+ TypeString(gossip_obj.Type)+ " signed by " + gossip_obj.Signer+".", http.StatusOK)
        return
    } else {
        //fmt.Println(util.GREEN+"Received new, valid", TypeString(gossip_obj.Type), "from "+util.GetSenderURL(r)+".", util.RESET)
        fmt.Println(util.GREEN,"Received new, valid ",TypeString(gossip_obj.Type), "signed by ",EntityString(gossip_obj.Signer), " at Period ",gossip_obj.Period," rega
        //fmt.Println(gossip_obj.Signature)
        ProcessValidObject(c, gossip_obj)
        c.SaveStorage()
    }
}
```



Concurrent Map Read and Write

```
Received new, valid STH_FRAG signed by Monitor 2 at Period 17 regarding(1 results) [8464/8878]  
Received new, valid REV_FRAG signed by UNKNOWN at Period 17 regarding CA 2 .  
fatal error: concurrent map read and map write  
Received new, valid REV_FRAG signed by Monitor 2 at Period 17 regarding CA 2 .  
  
goroutine 2396 [running]:
```

```
There already exists a REV_FULL Object [12929/14284]  
There already exists a REV_FULL Object  
Received new, valid STH_FRAG signed by Monitor 2 at Period 17 regarding Logger 2 .  
Connection failed to localhost:8083.  
Received new, valid STH_FRAG signed by UNKNOWN at Period 17 regarding Logger 2 .  
fatal error: concurrent map read and map write
```

- Bypassing the duplication check



Problem

Need thread safe implementation

→ synchronization primitives

Options:

1) `Sync.RWmutex`

A `RWMutex` is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer.

2) `Sync.Map` is optimized for two common use cases:

- a) when the entry for a given key is only ever written once but read many times, as in caches that only grow
- b) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys.

→ failed attempt on the old design

→ causing a deadlock or causing a significant performance drop which undermines the security assumption

Solution – Hybrid Approach

- 1) Move the threshold sign functionalities from the monitors to the gossipers

→ This means we only need to think about concurrency control on the gossipers

- 2) Use RWmutex to make sure Other goroutine's Read/ Write Access are blocked while we are try to store the data

- 3) Making sure the critical section is short

```
func (c *GossiperContext) StoreObject(o Gossip_object){
    switch o.Type{
    case STH,REV,ACC:
        c.RWlock.Lock()
        (*c.Storage_RAW)[o.Get_Counter_ID()] = o
        c.RWlock.Unlock()
    case CON:
        c.RWlock.Lock()
        (*c.Storage_RAW)[o.Get_Counter_ID()] = o
        (*c.Storage_POM_TEMP)[o.GetID()] = o
        c.RWlock.Unlock()
    case STH_FRAG,REV_FRAG,ACC_FRAG:
        c.RWlock.Lock()
        (*c.Storage_FRAG)[o.Get_Counter_ID()] = o
        c.RWlock.Unlock()
    case CON_FRAG:
        c.RWlock.Lock()
        (*c.Storage_FRAG)[o.Get_Counter_ID()] = o
        c.RWlock.Unlock()
    case STH_FULL,REV_FULL:
        c.RWlock.Lock()
        (*c.Storage_FULL)[o.GetID()] = o
        c.RWlock.Unlock()
    case ACC_FULL:
        c.RWlock.Lock()
        (*c.Storage_POM_TEMP)[o.GetID()] = o
        c.RWlock.Unlock()
    case CON_FULL:
        c.RWlock.Lock()
        (*c.Storage_POM)[o.GetID()] = o
        c.RWlock.Unlock()
    }
}
```



Monitors Each Period

- 1) Monitors query loggers/CAs for STH/REV information
- 2) Monitors send STH/REV (or ACC if Queried Loggers/CAs are not responding) to their connecting gossipers
- 3) Receives STH_FULL, REV_FULL, ACC_FULL(if applicable), CON_FULL (if applicable) from its gossipers
- 4) Prepare the Client update, store all the group authenticated data (File I/O, json encoding)

Note:

- 1) Gossiper-Monitor is a always pair
- 2) ACC will have Monitor's signature (Non-TSS) on it while STH/REV has Logger/CA's signature



Gossipers Each Period

- 1) Upon receiving any non-duplicate data, the Gossiper should first store it, invoke the data-specific handler and then gossip it to other gossipers it connects to
- 2) For any non-duplicate STH/REV received (with Logger/CA's signature, not yet signed by any gossipers), the gossipers should wait for Gossip_wait_time (A configurable constant that should be changed according to the information propagation delay in the network),
 - generate CON (Proof of misbehavior for Conflicting info) if a Conflicting STH/REV is received
 - Generate STH_FRAG/REV_FRAG (= sign the STH/REV with Gossiper's BLS partial signature) Otherwise
- 3) For any non-duplicate ACC/CON received, follow the same procedure as STH/REV except no conflict check is needed, because gossipers don't produce STH/REV/ACC themselves and CON has internal verification mechanism
- 4) For any non-duplicate STH_FRAG/REV_FRAG/ACC_FRAG/CON_FRAG received, the gossipers will increment its counter by 1 if the counter reaches the threshold while no fully signed version is yet stored, the gossipers should generate corresponding STH_FULL/REV_FULL/ACC_FRAG/CON_FRAG c and send it to the monitor (Also gossip it to other gossipers)



Sample Run

Topology: 3 loggers 3 CAs 4 monitors 4 gossipers

Running entities: Logger 1 , CA 1 , Monitor 1, Monitor 3, Gossiper 1, Gossiper 3

Malicious Entity: Logger 1 → sending conflicting STHs to Monitor 1 and Monitor 3

Unresponsive entities: Logger 2, Logger 3, CA 2, CA 3

Expected Output per period:

ACC_FULL = 4 (should stay at 4 because ACC_FULL is considered as temporary PoM and the cache will be wiped each period)

STH_FULL = 0 (should stay at 0 since the monitor will not query Loggers with PoM on file)

REV_FULL = 1 (+1 each period)

CON_FULL = 1 (should stay at 1)

Gossiper 3

```
testData > gossipdata > 3 > {} GossipStorage.Json > ...
18  "34": {
19    "Gossiper_URL": "localhost:8082",
20    "Period": "34",
21    "Num_sth": 1,
22    "Num_rev": 1,
23    "Num_acc": 4,
24    "Num_con": 2,
25    "Num_sth_frag": 0,
26    "Num_rev_frag": 2,
27    "Num_acc_frag": 2,
28    "Num_con_frag": 2,
29    "Num_STH_FULL": 0,
30    "Num_REV_FULL": 1,
31    "Num_ACC_FULL": 2,
32    "Num_CON_FULL": 1
33  },
34  "35": {
35    "Gossiper_URL": "localhost:8082",
36    "Period": "35",
37    "Num_sth": 1,
38    "Num_rev": 2,
39    "Num_acc": 8,
40    "Num_con": 2,
41    "Num_sth_frag": 0,
42    "Num_rev_frag": 4,
43    "Num_acc_frag": 4,
44    "Num_con_frag": 2,
45    "Num_STH_FULL": 0,
46    "Num_REV_FULL": 2,
47    "Num_ACC_FULL": 4,
48    "Num_CON_FULL": 1
49  }
```

Gossiper 1

```
testData > gossipdata > 1 > {} GossipStorage.Json > {} 35
18  "34": {
19    "Gossiper_URL": "localhost:8080",
20    "Period": "34",
21    "Num_sth": 1,
22    "Num_rev": 1,
23    "Num_acc": 4,
24    "Num_con": 2,
25    "Num_sth_frag": 0,
26    "Num_rev_frag": 2,
27    "Num_acc_frag": 4,
28    "Num_con_frag": 2,
29    "Num_STH_FULL": 0,
30    "Num_REV_FULL": 1,
31    "Num_ACC_FULL": 2,
32    "Num_CON_FULL": 1
33  },
34  "35": {
35    "Gossiper_URL": "localhost:8080",
36    "Period": "35",
37    "Num_sth": 1,
38    "Num_rev": 2,
39    "Num_acc": 8,
40    "Num_con": 2,
41    "Num_sth_frag": 0,
42    "Num_rev_frag": 4,
43    "Num_acc_frag": 8,
44    "Num_con_frag": 2,
45    "Num_STH_FULL": 0,
46    "Num_REV_FULL": 2,
47    "Num_ACC_FULL": 4,
48    "Num_CON_FULL": 1
49  }
```

testData > monitordata > 1 > {} CONFLICT_POM.json > ...

```
You, 1 second ago | 1 author (You)
1  [ You, last week • Major Update ...
2
3  {
4    "application": "CTng",
5    "period": "0",
6    "type": "http://ctng.uconn.edu/304",
7    "signer": "",
8    "signers": {
9      "0": "localhost:8082",
10     "1": "localhost:8080"
11   },
12   "signature": [
13     "{\n\"sign\": \"1889007b7a6c5cea7b678e577c77669857cc831fa52f62aea4a11d129a17d776067f7eea1a5d85930343e04d95d67f0b\", \"ids\": [\"localhost:8080\", \"localhost:8082\"]\",
14     \"\"
15   ],
16   "timestamp": "2022-11-26T19:47:01Z",
17   "crypto_scheme": "BLS",
18   "payload": [
19     "localhost:9000",
20     "localhost:9000{\n\"Timestamp\": \"47\", \"RootHash\": \"f29ce321bae5e5b8326bca5f5aa4a3b03ffbaa1353f23606d763a7b978d05119\", \"TreeSize\": 0}",
21     "localhost:9000{\n\"Timestamp\": \"47\", \"RootHash\": \"b31149594a476f33e0c3aed063a111673ab5ae16c5d49c9b84cc365ea623a4cc\", \"TreeSize\": 0}"
22   ]
23 }
```

Monitor 1:
CON_FULL

testData > monitordata > 3 > {} CONFLICT_POM.json > ...

```
You, 1 second ago | 1 author (You)
1  [ You, last week • Major Update ...
2
3  {
4    "application": "CTng",
5    "period": "0",
6    "type": "http://ctng.uconn.edu/304",
7    "signer": "",
8    "signers": {
9      "0": "localhost:8082",
10     "1": "localhost:8080"
11   },
12   "signature": [
13     "{\n\"sign\": \"1889007b7a6c5cea7b678e577c77669857cc831fa52f62aea4a11d129a17d776067f7eea1a5d85930343e04d95d67f0b\", \"ids\": [\"localhost:8080\", \"localhost:8082\"]\",
14     \"\"
15   ],
16   "timestamp": "2022-11-26T19:47:01Z",
17   "crypto_scheme": "BLS",
18   "payload": [
19     "localhost:9000",
20     "localhost:9000{\n\"Timestamp\": \"47\", \"RootHash\": \"f29ce321bae5e5b8326bca5f5aa4a3b03ffbaa1353f23606d763a7b978d05119\", \"TreeSize\": 0}",
21     "localhost:9000{\n\"Timestamp\": \"47\", \"RootHash\": \"b31149594a476f33e0c3aed063a111673ab5ae16c5d49c9b84cc365ea623a4cc\", \"TreeSize\": 0}"
22   ]
23 }
```

Monitor 3:
CON_FULL



Sources

[Certificate of Authenticity for the 100% Japanese Wagyu Beef served at Greystone - Wagyu Beef | San Diego Steakhouse | Best in Gaslamp \(greystonesteakhouse.com\)](#)

[SSL Precertificates: What They Are & What They Do \(thesslstore.com\)](#)

[How CT Works : Certificate Transparency](#)

[Golang | Goroutine vs Thread - GeeksforGeeks](#)

[sync package - sync - Go Packages](#)

[time.AfterFunc\(\) Function in Golang With Examples - GeeksforGeeks](#)

[os package - os - Go Packages](#)