Waku Core
Contributors

Making Peer-To-Peer Networks Reliable

Agenda

1. Using libp2p in end user devices
2. SDS: End-to-end reliability protocol
3. Waku Sync protocol

End User Devices

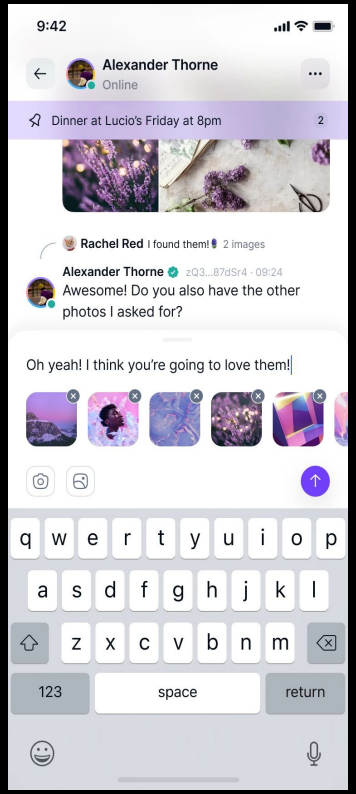
Learning from
–

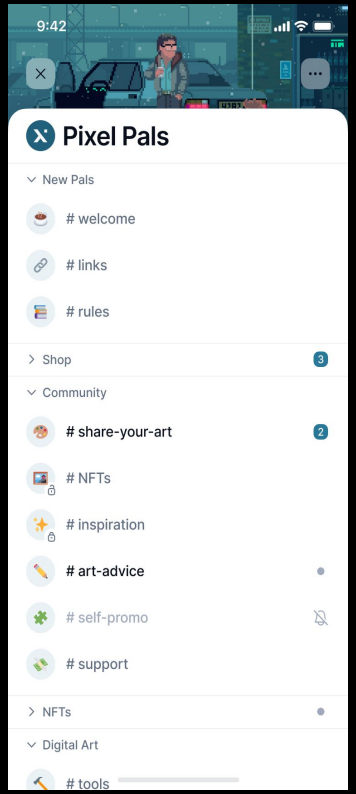
Status: Chat app

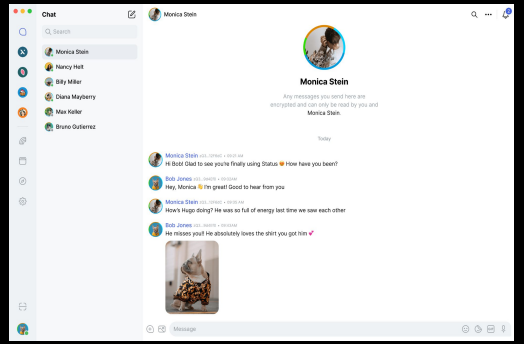
- One-to-one
- Discord-like
- Mobile
- Desktop

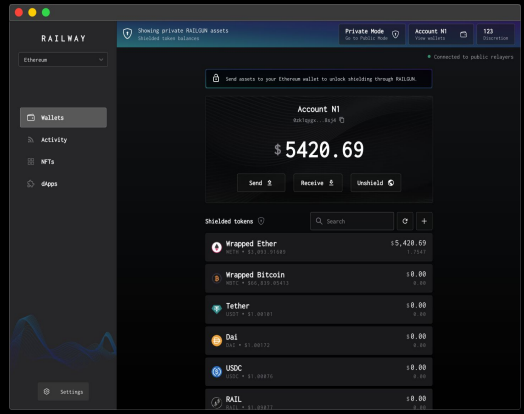
RAILGUN & Others

- Browser
- Mobile
- Desktop
- Node (cloud)









Whisper Lessons

Whisper devp2p floodsub

→ Libp2p gossipsub (Waku Relay)

- Discv5, DNS Discovery

Light protocols (request-response)

- Store: Retrieve missed messages
- Light push: Send message with ACK
- Filter: subscribe to subset of messages
- Peer exchange: get peers
- RLN Relay: bandwidth capping

Data Sync - end-to-end/application reliability

- MVDS
- Ack based

Initial Setup

Desktop

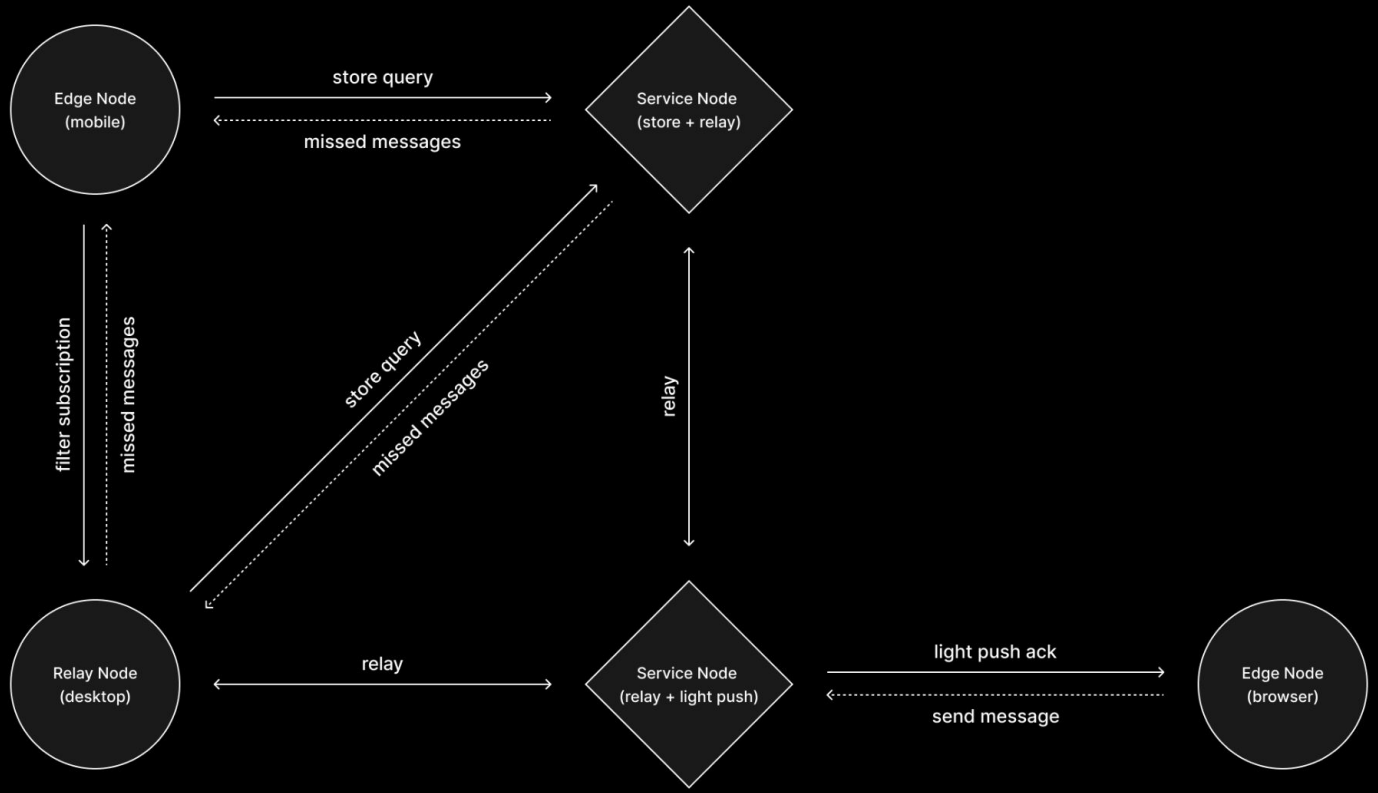
- Relay node
- Provides services to mobile (light protocols)

Mobile & Browser

- Libp2p-native
- No REST API/Web Gateway
- Edge node
- Use light protocols

Sharding

- Split gossipsub traffic


Edge Node
(mobile)
store query
missed messages
Service Node
(store + relay)
filter subscription
missed messages
store query
missed messages
relay
Relay Node
(desktop)
relay
Service Node
(relay + light push)
light push ack
send message
Edge Node
(browser)


Latest Learnings

Latest Learnings

Gossipsub: *Am I online? Did you get my message?*

- 10-30s range (TCP timeout)
- End user impact
- Micro disconnections

Latest Learnings

Desktop: *Poof, I'm gone*

- ENR time scale (Discv5 and Waku peer exchange)
- Finding online nodes
- Reconnecting to online nodes

Latest Learnings

DNS not always reliable

- OS controlled on mobile

WebSocket = 

- Long connections

Other Learnings

Ack-based data sync

- Does not scale: 100-1000 users traffic amplification


Solutions (so far)

Solutions

Ping galore

- Random subset of gossipsub peers
- Gives better heuristics
  - Is it you or me?
- Light protocol pings
  - *Am I still subscribed?*

Solutions

Redundancy for light protocols

- Replicate gossipsub’s
- Not for store

Waku Peer Exchange + Discv5 on Desktop

- Faster bootstrap
- Discv5 still used for wider array of peers

Solutions

Periodic store checks

- Time range query
- Only retrieve message hashes
- Helps with micro disconnections
- Remembers last successful check
- Potential ramp down with e2e reliability and Waku Sync

Store message confirmation

- Wait & ask store node if message seen
    - Store populates from relay (gossipsub)

Check the clock

- Was I suspended?


What’s Next?

What's Next?

Scalable end-to-end reliability protocol (data sync)

Waku Sync

Clean API

Decentralized Store

Bandwidth Impact

Questions

Scalable Data Sync Protocol

A Brief Introduction

SDS Protocol

–

Introduction

- Scalable Data Synchronisation
- App-level e2e reliability (over gossipsub)
- Group communications over p2p transport
- Model: consolidate distributed logs
- First application: group chat
- Aims for (partial) causal ordering and eventual consistency
- Designed for large-scale, dynamic participant groups; works for 1:1 case

SDS Protocol

–

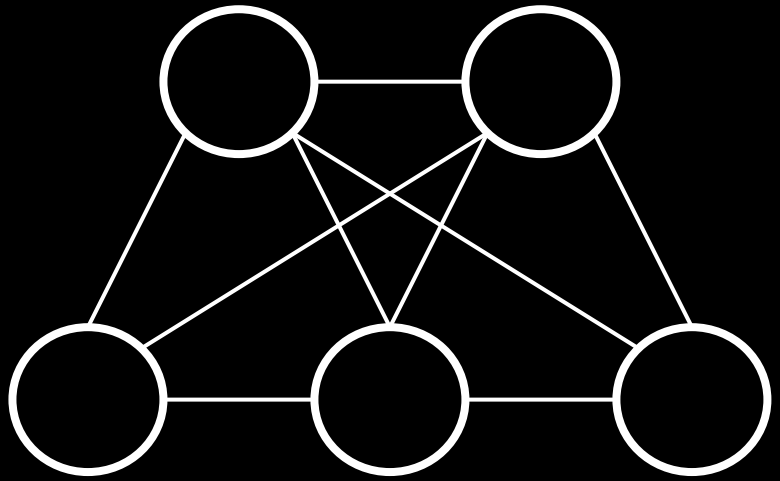
Scalability Challenges

- Direct interaction in 1:1 chats and small groups:
  - Simple, direct ACKs and sync requests for missing messages
  - Minimal resource load—only two nodes involved, even over broadcast-type transport
- Challenges in large groups:
  - High broadcast traffic for an explosion of ACKs and other “direct” interactions in larger groups.
  - Increased redundancy and resource usage.
- Risk of network flooding

SDS Protocol

–

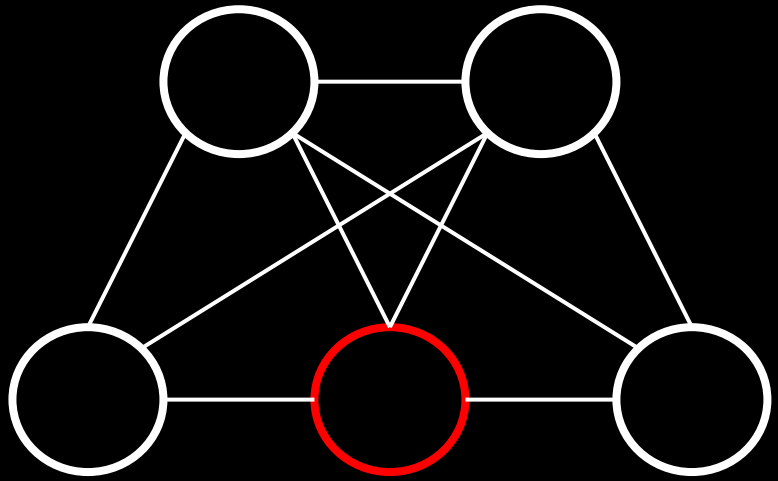
Scalability challenges



SDS Protocol

–

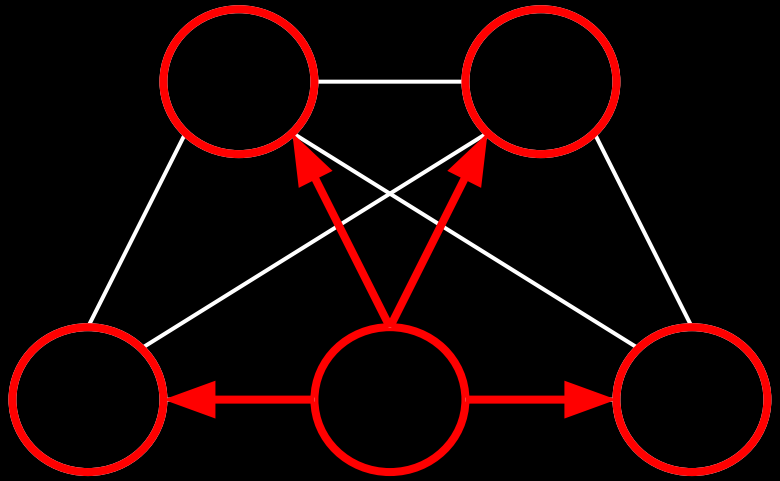
Scalability challenges



SDS Protocol

–

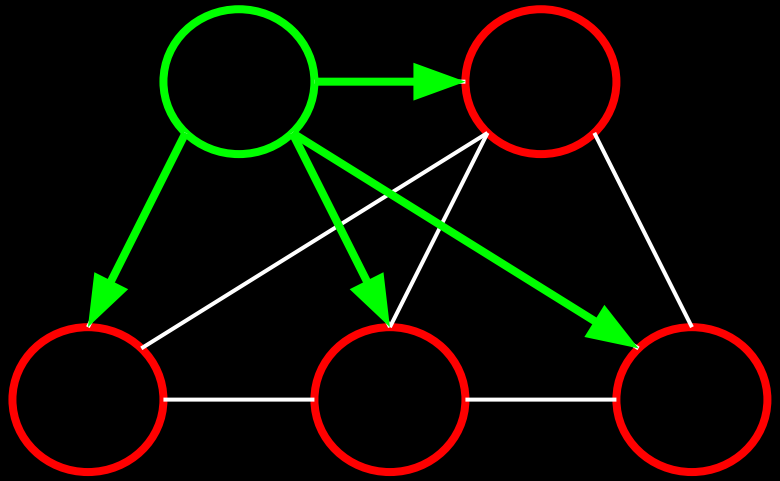
Scalability challenges



SDS Protocol

–

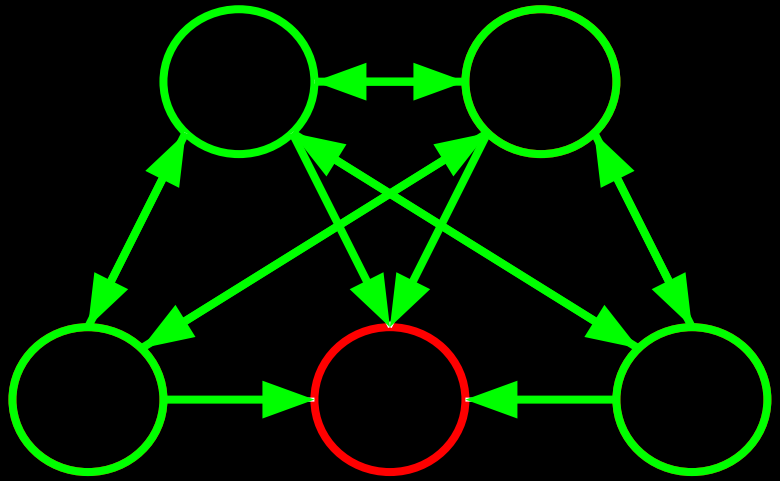
Scalability challenges



SDS Protocol

–

Scalability challenges



SDS Protocol

–

Key Goals

- Partial causal ordering, eventual consistency
- Piggyback on existing messages, if possible
- Participant responsibilities:
  - Push: Infer (at least partial) ACK
  - Pull: Determine gaps in causal history
- Limit redundancy and latency:
  - Prefer *eager push* to propagate new messages quickly
  - Trigger *lazy pull* only to fill in gaps and retrieve missing messages

SDS Protocol

–

Main protocol features

- Causal ordering via Lamport timestamps
- Causal dependency acyclic graph to detect missing messages
- Bloom filters track message acknowledgments
- Dependency checks, ACK checks and conflict resolution on delivery

Message Payload	“Hi, Group!”
Message ID	ID_D
Lamport Timestamp	4
Partial Causal History	ID_C, ID_B, ID_A
Bloom filter	101101011101

SDS Protocol

–

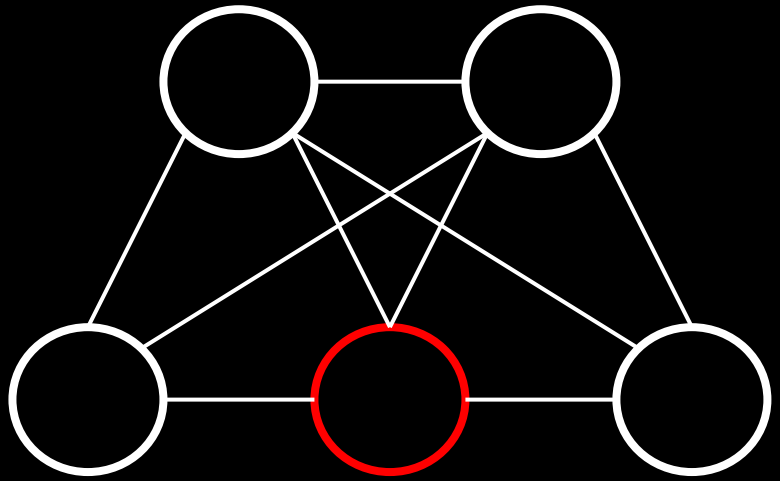
Scalable acknowledgements

- Published messages (via Waku Relay) are considered unacknowledged
- Periodic rebroadcast until ACK (eager push)
- Ways to ACK a message:
  - Inclusion in received causal DAG
  - Probabilistic ACK with bloom filters

SDS Protocol

–

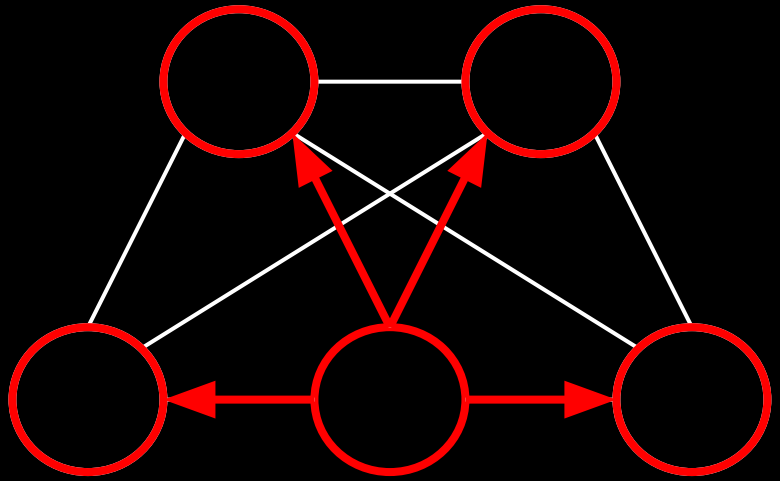
Scalable acknowledgements



SDS Protocol

–

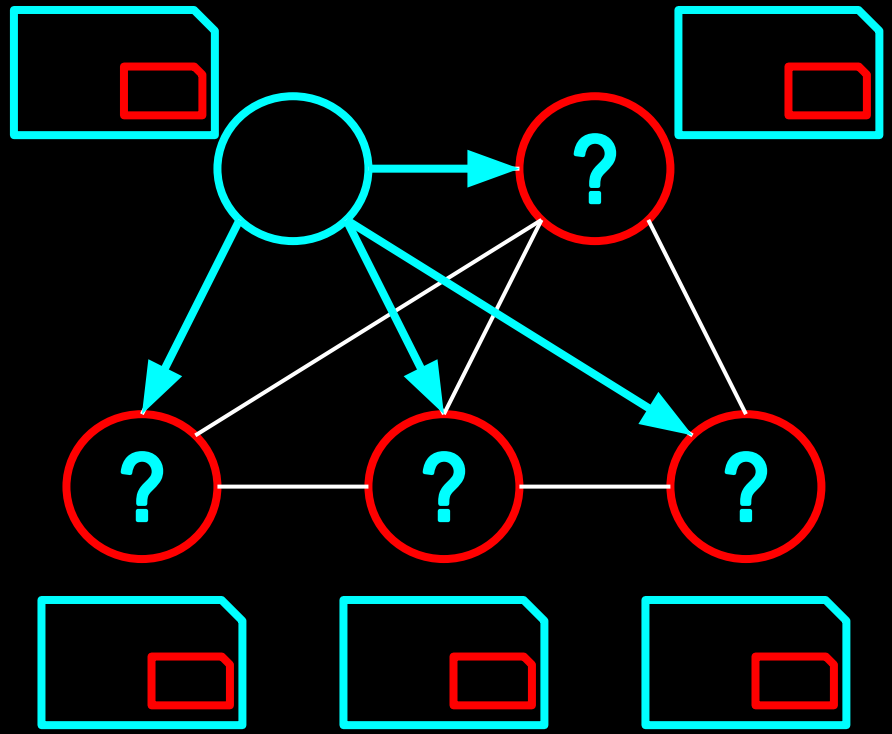
Scalable acknowledgements



SDS Protocol

–

Scalable acknowledgements



SDS Protocol

–

Causal Ordering and Dependencies

- Causal ordering:
  - Ordering based on **Lamport timestamps**.
  - Guarantees consistent order for all participants.
- Causal dependencies:
  - Each message includes IDs of prior messages (its **causal history**).
  - Ensures delivery order respects message relationships.
- Handling unmet dependencies:
  - Messages with unmet dependencies are buffered.
  - Missing dependencies can be *lazily* fetched as needed.

SDS Protocol

–

Periodic Syncs and Buffer Sweeps

- Periodic sync messages to maintain state
- Outgoing and incoming buffer sweeps
- Support for ephemeral messages without reliability overhead

SDS Protocol

–

Conclusion

- Separation of concerns: gossipsub for broadcast, SDS for reliability
- App-level reliability raises scalability concerns
- Waku leverages SDS for partial causal ordering and eventual consistency
- SDS offers tunable scalability

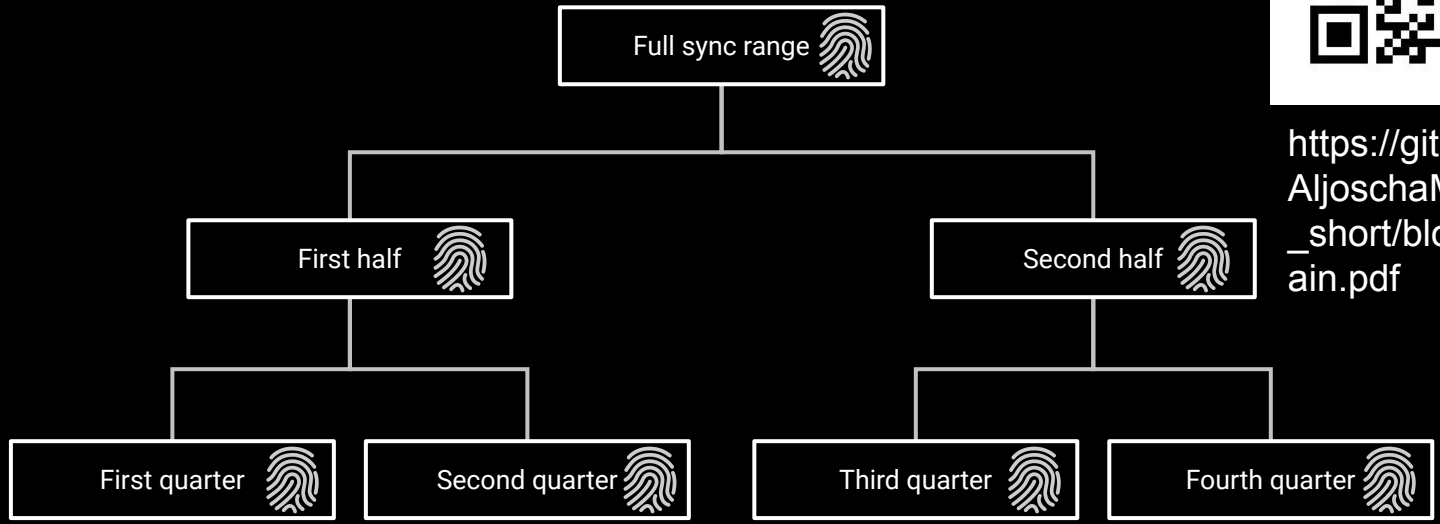
Waku Sync Protocol

Waku Sync

–

RBSR overview

Range Based Set Reconciliation
- Total order
- Range fingerprint fonction



https://github.com/AljoschaMeyer/rbsr_short/blob/main/main.pdf

Waku Sync

–

RBSR in Waku

Waku Messages

- Timestamp
- Hash
- RLN proofs

Knowing the context allow us to optimize.

- Range selection
- Fingerprint function
- Storage impl.

Waku Sync

–

Version 1.0

Negentropy protocol

- Data requirements matched by Waku messages
- Efficient tree storage
- C++ implementation
- Waku Nim wrapper

What is wrong?

- Pruning inefficiencies
- Peer asymmetry
- One more dependency

https://github.com/hoytech/negentropy

Waku Sync

–

Version 2.0

In house RBSR implementation

- Range selection
- Fingerprint function
- Storage
- Peer symmetry

Design, impl. and testing takes time!

https://github.com/waku-org/research/issues/102

Waku Sync

–

What’s next?

Waku Sync can propagate messages in the network.
It can be used in conjunction with other protocols.

- Light push
- Filter
- Relay

Maybe even replace them?

Details

Franck
https://x.com/fryorcraken

Hanno
https://x.com/4aelius

Simon-Pierre
https://x.com/SionoisRP

https://waku.org
https://github.com/waku-org/specs/