Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication

Yang Wang, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin
Resolving a long-standing trade-off

- **Efficiency**
  - Write to f+1 nodes and read from 1 node

- **Availability**
  - Aggressive timeout for failure detection

- **Consistency**
  - Read always returns the data of the latest write.
Resolving a long-standing trade-off

• Efficiency
  – Write to f+1 nodes and read from 1 node

• Availability
  – Aggressive timeout for failure detection

• Consistency
  – Read always returns the data of the latest write.
Resolving a long-standing trade-off

• Efficiency
  – Write to f+1 nodes and read from 1 node

• Availability
  – Aggressive timeout for failure detection

• Consistency
  – Read always returns the data of the latest write.
Gnothi Overview

Gnothi resolves the trade-off ...

... but only for block storage, meaning ...

– A fixed number of fixed-size blocks.
– A request reads/writes a single block.

Key ideas:

– Don’t insist that nodes have identical state.
– A node knows which blocks are fresh/stale.

Gnothi Seauton – Know yourself
Separating Data and Metadata

Metadata: blockNo, client ID, ...

Data

Write request

LAN

Clients

2f+1 nodes

Metadata Size: 24 bytes for a block (4K to 1M)
Rest of the talk

• Why is the trade-off challenging?

• How does Gnothi resolve the trade-off?

• How well does Gnothi perform?
Why is the trade-off challenging?

How to handle a timeout?

Synchronous Primary Backup
(Remus, HBase, Hypervisor, …)

1

Continue with 1 node
Use conservative timeout

Timeout

Asynchronous Replication
(Paxos, …)

1

Send to 2f+1 nodes and waits for f+1 ACKs

Timeout

• Can we have both f+1 replication and short timeout?
Why is the trade-off challenging?

× Continue with 1 node? – Not safe.
 ✓ Switch to another node. (Cheap Paxos, ZZ, ...)
?

However, state of newly enlisted node may be incomplete.
  – One solution: on switch, copy all data to new node – bad availability.
Rest of the talk

• Why is the trade-off challenging?

• How does Gnothi resolve the trade-off?

• How well does Gnothi perform?
Gnothi: Nodes can be incomplete

- A new write will overwrite the block anyway.
- Read can be processed correctly
  - As long as a node knows which blocks are stale
- Recovery can be processed correctly
  - As long as a node knows which block is the latest one

![Diagram of block operations](image)
How does Gnothi work?

How to perform writes and reads efficiently when no failures occur?
  - Write to f+1 and read from 1

How to continue processing requests during failures?
  - Still write to f+1 and read from 1

How to recover the failed node efficiently?
How to perform writes and reads efficiently when no failures occur?

- Data replicated $f+1$ times

Maintain a single bit for each block: “do I have the current data?”

Metadata ensures read can be processed correctly.
Load-balanced Data Distribution

Gnothi Block Drivers

LAN

Slice 1 | Slice 2 | Slice 3
---|---|---

Divide space into multiple slices

Preferred Storage | Reserve Storage
---|---
Slice 1 | Slice 2 | 3
Slice 2 | Slice 3 | 1
Slice 1 | Slice 3 | 2

Node 1 | Node 2 | Node 3
---|---|---

Evenly distribute slices to different preferred nodes
Load-balanced Data Distribution

Divide space into multiple slices

Gnothi Block Drivers

LAN

Preferred Storage

Reserve Storage

Node 1

Node 2

Node 3

Evenly distribute slices to different preferred nodes
How to continue processing requests during failures?

• Metadata replicated 2f+1 times

Metadata allows a node to process requests correctly.
Catch-up problem in recovery

• Recovery speed vs Execution speed
  – Traditional systems have the catch-up problem

Node 1

Node 2

Node 3

Can I catch up?
How to recover the failed node efficiently?

• Separate metadata and data recovery
  – Phase 1: Metadata recovery – fast
How to recover the failed node efficiently?

• Separate metadata and data recovery
  – Phase 1: Metadata recovery – fast
  – Phase 2: Data recovery – slow, in background
Rest of the talk

• Why is the trade-off challenging?

• How does Gnothi resolve the trade-off?

• How well does Gnothi perform?
Evaluation

• Throughput
  – Compare to a Gaios (Bolosky et al. NSDI 2011) like system G’.
  – Sequential/Random read/write
  – f=1 (Gnothi-3, G’-3) and f=2 (Gnothi-5 and G’-5)
  – Block size 4K, 64K, and 1M

• Failure Recovery
  – Compare Gnothi to G’ and Cheap Paxos
  – How long does recovery take?
  – What is the client throughput during recovery?
Gnothi can achieve 40%-64% more write throughput and scalable read throughput.
Higher throughput during recovery

Gnothi does not block long for failures.
Gnothi can achieve 100%-200% more throughput during recovery.
Gnothi can always catch up

Tunable recovery speed.
In Gnothi, the recovering node can always catch up with others.
Conclusion

• Separate Data and Metadata
  – Replication
    • Improve efficiency.
    • Ensure availability and consistency during failures.
  – Recovery
    • Ensure catch-up.