Fault Tolerance via the State Machine Replication Approach

Favian Contreras

Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial

Written by Fred Schneider

Why a Tutorial?

The “State Machine Approach” was introduced by Leslie Lamport in “Time, Clocks and Ordering of Events in Distributed Systems.”

Problem

Data storage needs to be able to tolerate faults!

How do we do this?

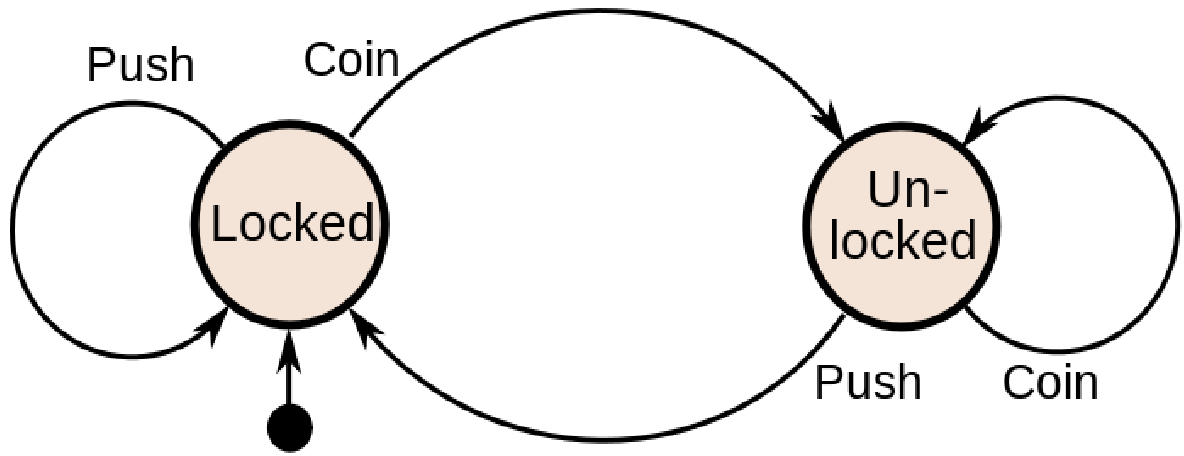
Replicate data in a smart and efficient way!!!

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

State Machines

- State Variables
- Deterministic Commands



Requests and Causality, Happens Before Tutorial

- Process order consistent with potentially causality.
- Client A sends r, then r'.
- r is processed before r'.
- r causes Client B to send r'.
- r is processed before r'.

State Machine Coding

- State Machines are procedures
- Client calls procedure
- Avoid loops.
- More flexible structure.

Consensus

- Termination
- Validity
- Integrity
- Agreement

- Ensures procedures are called in same order across all machines

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Faults

- Byzantine Faults:
  - Malicious/arbitrary behavior by faulty components.
  - Weakest possible failure assumption.
- Fail-Stop Faults:
  - Changes to fail state and stops.
- Crash Faults:
  - Not mentioned in tutorial.
  - It is an omission failure, similar to fail-stop

Tolerating Faults

- t fault tolerant
  - ≤ t components become faulty
  - Simply where the guarantees end.
- Statistical Measures
  - Mean time between failures
  - Probability of failure over interval
  - other

Tolerating Faults

- t fault tolerant
  - $\leq t$ components become faulty
  - Simply where the guarantees end.
- Statistical Measures
  - Mean time between failures
  - Probability of failure over interval
  - other

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Fault Tolerant State Machines

- Implement the state machine on multiple processors.
- State Machine Replication
  - Each starts in the same initial state
  - Executes the same requests
  - Requires consensus to execute in same order
  - Deterministic, each will do the exact same thing
  - Produce the same output.

t Fault-Tolerance

- Replicas need to be coordinated
- Replica coordination:
  - Agreement:
    - Every non-faulty replica receives every request.
  - Order:
    - Every non-faulty replica processes the requests in the same relative order.

t Fault-Tolerance

- Byzantine Faults:
  - How many replicas needed in general?
  - Why?
- Fail-Stop Faults:
  - How many replicas needed in general?
  - Why?

Outline

- State machines
- Faults
- State Machine Replication
  - Agreement
  - Ordering
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Agreement

- "The transmitter" disseminates a value, then:
  - IC1: All non-faulty processors agree on the same value
  - IC2: If transmitter is non-faulty, agree on its value.
- Client can
  - be the transmitter
  - send request to one replica, who is transmitter

Outline

- State machines
- Faults
- State Machine Replication
  - Agreement
  - Ordering
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Ordering

- Unique identifier, uid on each request
- Total ordering on uid.
- Request, r is stable if
  - Cannot receive request with uid(r') < uid(r)
- Process a request once it is stable.
- Logical clocks can be the basis for unique id.
- Stability tests for logical clocks?
  - Byzantine faults?

Ordering

- Can use synchronized real-time clocks.
- Max one request at every tick.
- If clocks synchronized within δ,
  - Message delay > δ
- Stability tests?
- Potential Problems?
  - State Machine lag behind clients by Δ (test 1)
  - Never passed on crash failures (test 2)

More Ordering...

- Can the replicas generate uid's?
- Of course!
- Consensus is the key!
- State machines propose candidate id's.
- One of these selected, becomes unique id.

Constraints

- UID1: $cuid(sm_i, r) <= uid(r)$.
- UID2: If a request r' is seen by sm_i after r has been accepted by sm_i, then $uid(r') < cuid(sm_i, r')$.

How to generate uid's?

- Requirements:
  - UID1 and UID2 be satisfied
  - r != r' $\Rightarrow$ *uid(r) != uid(r')*
  - Every request seen is eventually accepted.
- Define:
  - SEEN(i) = largest $cuid(sm_i,r)$ *assigned to any request so far seen at* sm_i
  - *ACCEPT(i)* = largest $cuid(sm_i,r)$ *assigned to any request so far accepted by* sm_i

Generating uid's....

- $cuid(sm_i,r) = max(SEEN(i), ACCEPT(i)) + 1 + i/N$.
- $uid(r) = max(cuid(sm_i,r))$
- Stability test?
- Potential Problems?
  - Could affect causality of requests
  - Client does not communicate until request is accepted.
- More or less communication needed?

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Tolerating failures

- Failed output device or voter:
  - Replicate?
  - Use physical properties to tolerate failures, like the flaps example in the paper.
  - Add enough redundancy in fail-stop systems
- Client Failure:
  - Who cares?
  - If sharing processor, use that SM

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Reconfiguration

- Would removing failed systems help us tolerate more faults?
- Yes, it seems!
- P(t) = total processor at time t
- F(t) = Failed Processors at time t
- Assume Combine function, P(t) – F(t) > Enuf
- Enuf = P(t)/2 for byzantine failures
- Enuf = 0 for fail-stop.

Reconfiguration

- F1: If Byzantine failures, then faulty machines are removed from the system before combining function is violated.
- F2: In any case, repaired processors are added before combining function is violated.
- Might actually improve system performance.
- Fewer messages, faster consensus.

Integrating repaired objects

- Element must be non-faulty and must have the current state before it can proceed.
- If it is a replica, and failure is fail-stop:
  - Receive a checkpoint/state from another replica.
  - Forward messages, until it gets the ordered messages from client.
- Byzantine fault?

Discussion

- Why does any of this matter?
- What is the best case scenario in terms of replications for fault tolerance?
- Is the state machine approach still feasible?
- Are there any other ways to handle BFT?
- Which was the most interesting?

Takeaways

- The State Machine approach is flexible.
- Replication with consensus, given deterministic machines, provides fault tolerance.
- Depending on assumptions, may need more replications, may use different strategies.

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Reconfiguring
- Chain Replication

Chain Replication For Supporting High Throughput and Availability

- Robert Van Renesse
- Fred Schneider

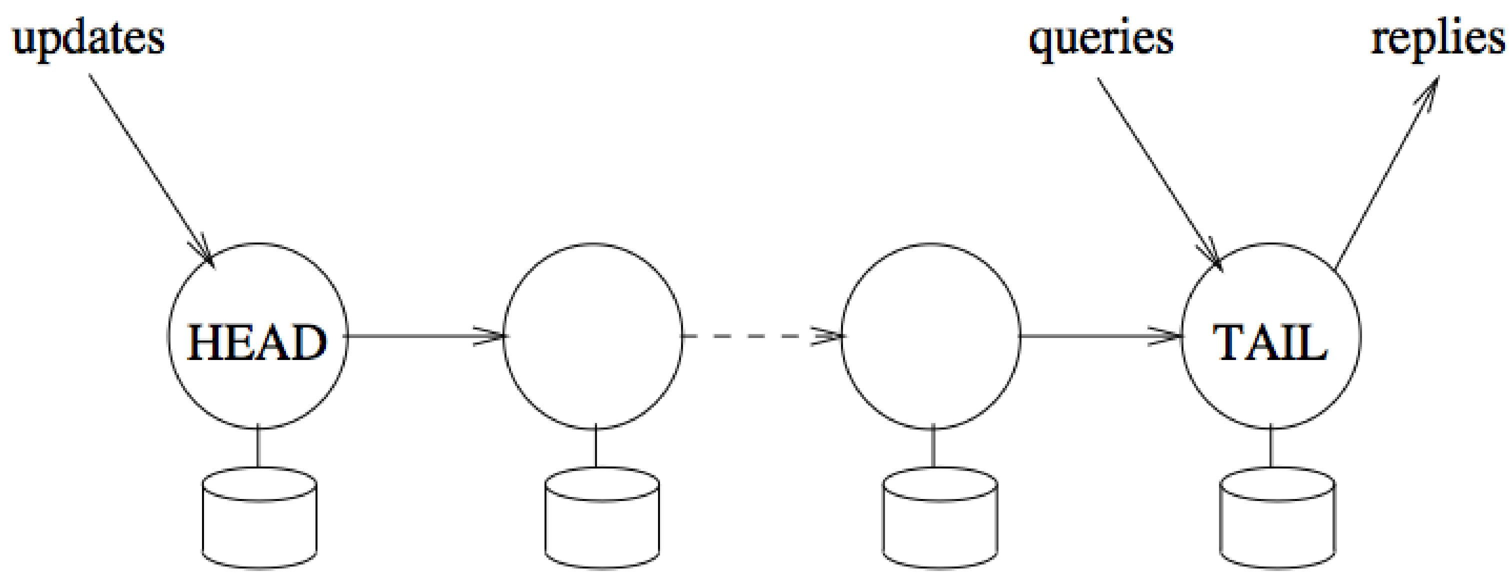
Primary-Backup

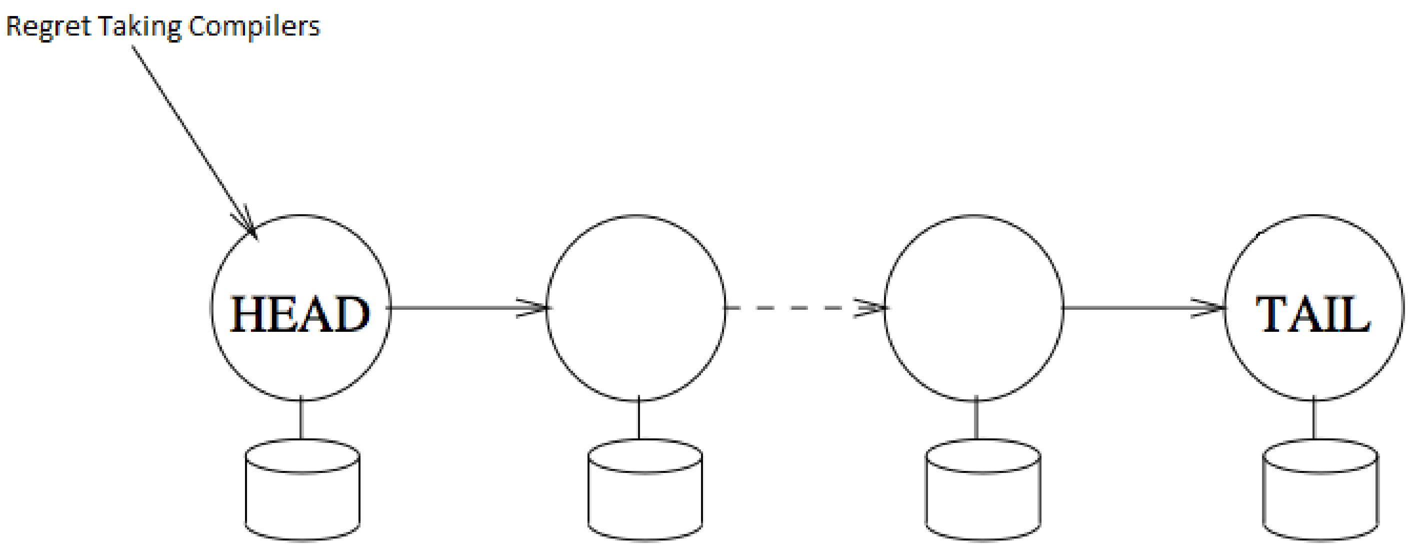
- Different from State Machine Replication?
- Serial version of State Machine Replication
- Only the primary does the processing
- Updates sent to the backups.

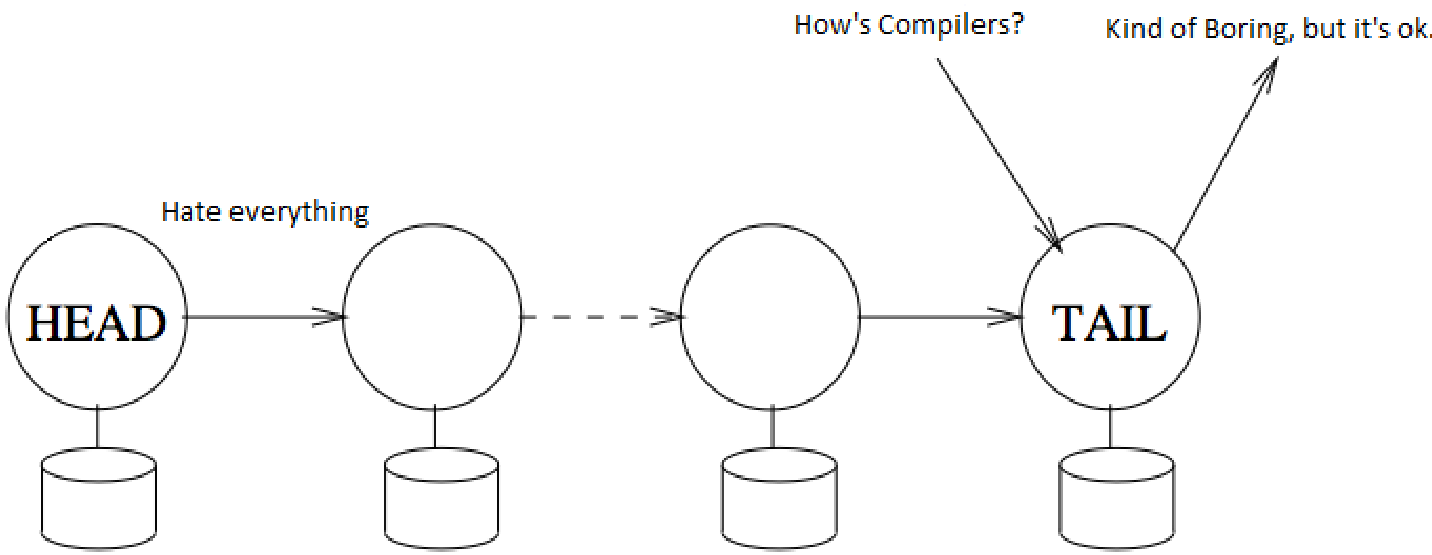
Chain Replication Assumes:

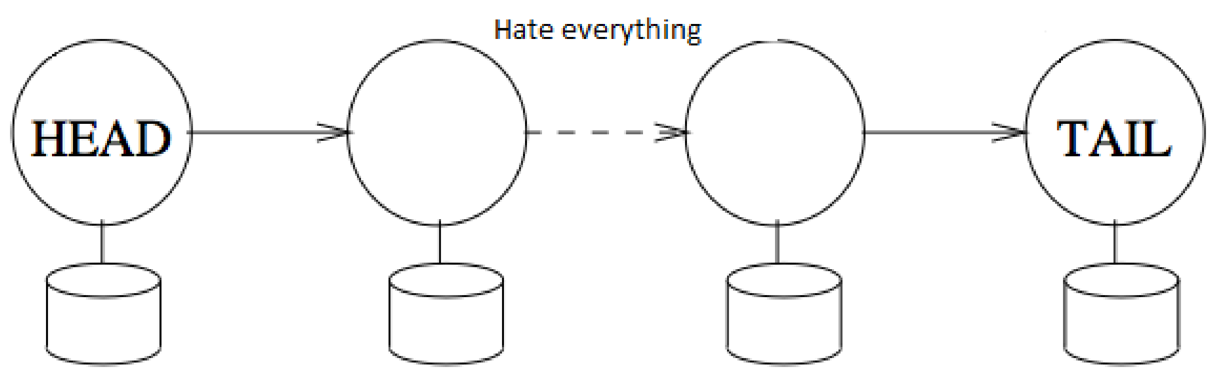
- No partition tolerance.
- Chain replication: Consistency, availability.
- A partitioned server == failed server.
- High Throughput.
- Fail-stop processors.
- A universally accessible, failure resistant or replicated Master, which can detect failures.

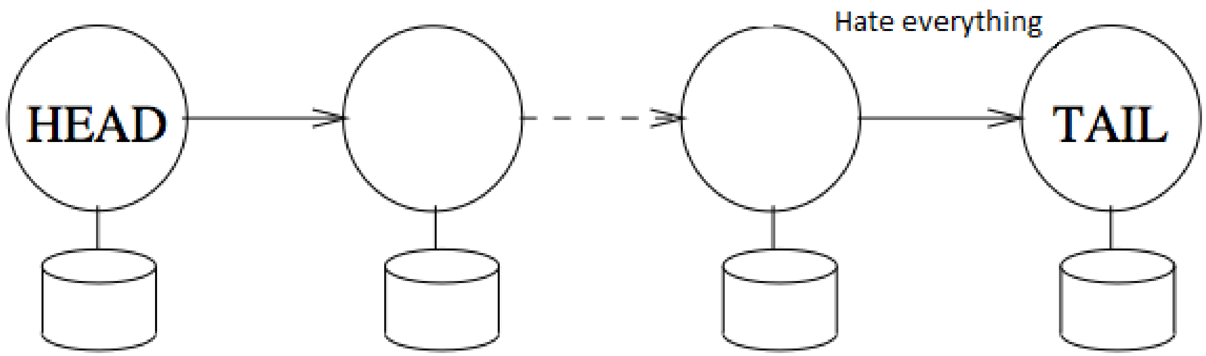
Serial State Machine Replication

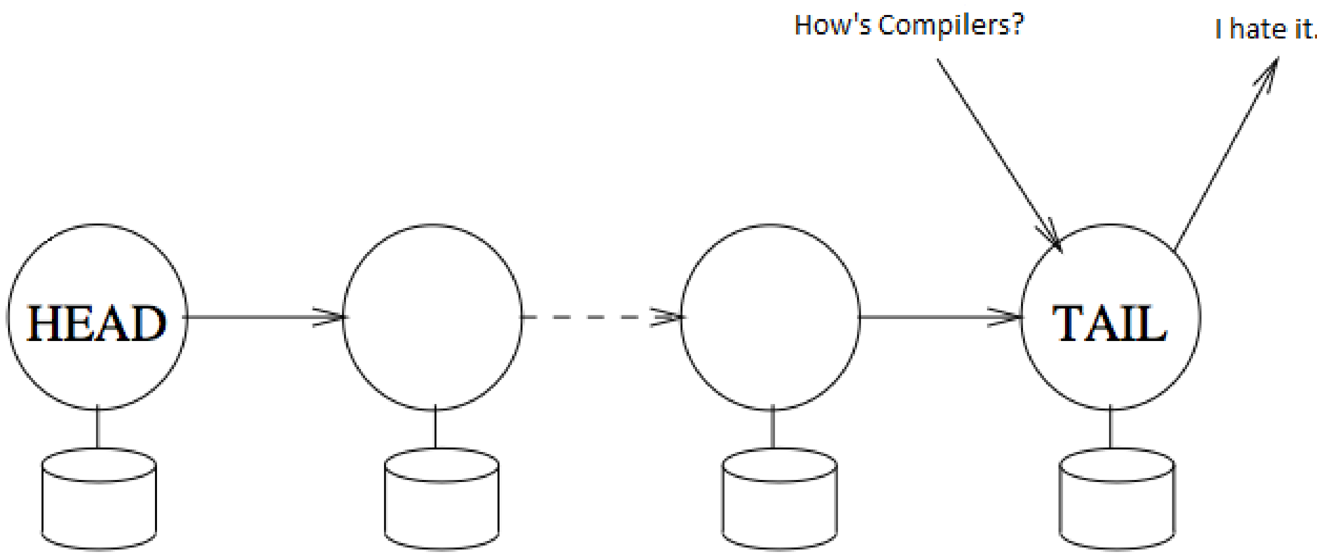



Regret Taking Compilers
HEAD
TAIL


How's Compilers?
Kind of Boring, but it's ok.
Hate everything
HEAD
TAIL


Hate everything
HEAD
TAIL


Hate everything
HEAD
TAIL


How's Compilers?
I hate it.
HEAD
TAIL

Reads and Writes

- Reads go to any non-faulty tail.
  - Just tail, 1 server per chain
- Writes propagate through all non-faulty servers.
  - t-1 severs per chain

Master!!

- Assumed to never fail or replicated w/ Paxos
- Head fails?
- Tail fails?
- Other fails?

Sources

- Fred Schneider photo: http://www.cs.cornell.edu/~caruana/web.pictures/pages/fred.schneider.sailing.c%26c.htm
- Robert van Renesse photo: http://www.cs.cornell.edu/annual_report/00-01/bios.htm
- Most Slides: Hari Shreedharan, http://www.cs.cornell.edu/Courses/CS6410/2009fa/lectures/23-replication.pdf
- State Machine photo: http://upload.wikimedia.org/wikipedia/commons/9/9e/Turnstile_state_machine_colored.svg

Extras!!!

Storage Systems

- Store objects.
- Query existing objects.
- Update existing objects.
- Usually offers strong consistency guarantees.
- Request processed based on some order.
- Effect of updates reflected in subsequent queries.

Handling failures

- Failures are detected by God/Master.
- On detecting failure, Master:
  - informs its predecessor or successor in the chain
  - informs each node its new neighbors
- Clients ask the master for information regarding the head and the tail.

Adding a new replica

- Current tail, T notified it is no longer the tail.
- State, Un-ACK-ed requests now transmitted to the new tail.
- Master notified of the new tail.
- Clients notified of new tail.

Unavailability

- Head failure:
  - Query processing uninterrupted,
  - update processing unavailable till new head takes on responsibility.
- Middle failure:
  - Query processing uninterrupted,
  - update processing might be delayed.
- Tail failure:
  - Query and update processing unavailable, until new tail takes over.