Replication

Hari Shreedharan

Papers!

- *Implementing Fault-Tolerant Services using the State Machine Approach (Dec, 1990)*

Fred B. Schneider
Cornell University.

Who introduced it?

- Most people confused about who introduced the “State Machine Approach.”
- Introduced by Leslie Lamport in the seminal “Time, Clocks and Ordering of events in distributed systems.”

Whose idea?

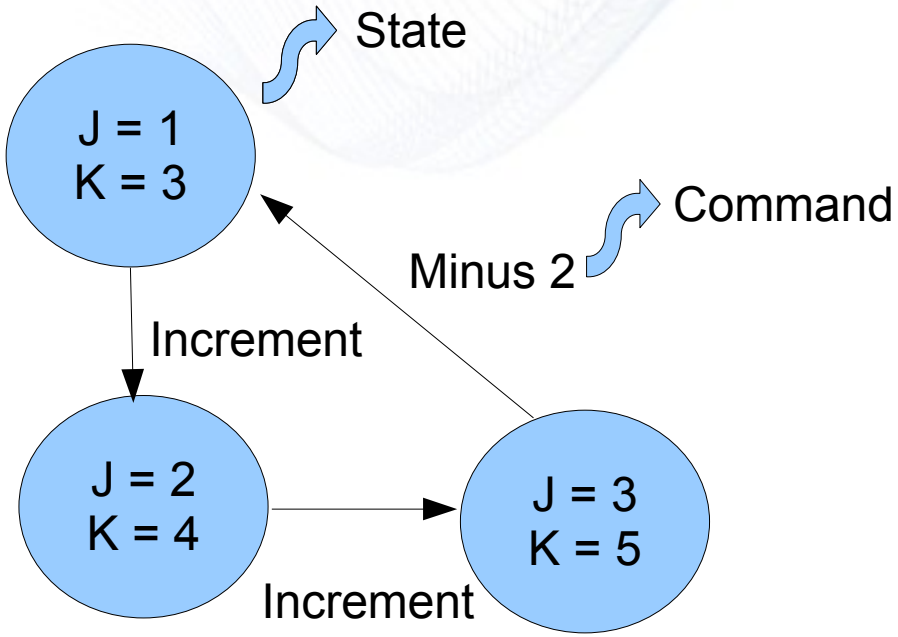
- Lamport says on his page, about that paper: *"This is my most often cited paper. Many computer scientists claim to have read it. But I have rarely encountered anyone who was aware that the paper said anything about state machines. People seem to think that it is about either the causality relation on events in a distributed system, or the distributed mutual exclusion problem. People have insisted that there is nothing about state machines in the paper. I've even had to go back and reread it to convince myself that I really did remember what I had written."*
- So Lamport introduced it, Schneider surveyed it :)

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- Chain Replication

State Machines

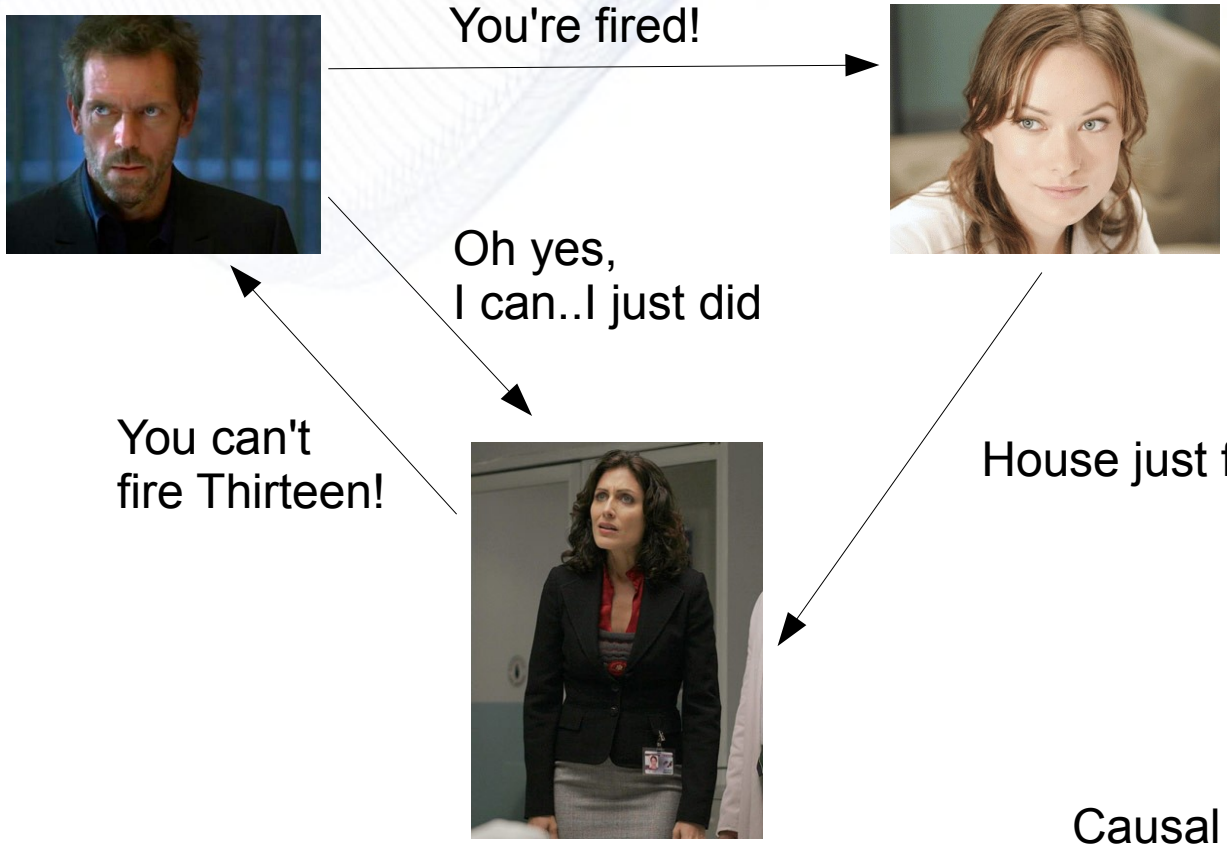
- State variables
- Deterministic Commands




```
memory : state_machine
    var store : array [0 . . n] of word

    read : command(loc : 0 . . n)
            send store[loc] to client
            end read;

    write : command(loc : 0 . . n, value :
    word)
            store[loc] := value
            end write;

    end memory
```

Requests and Causality!

- Commands processed in potentially causal order.



Causal Ordering is important!

More formally...

- O1 : Requests from a single client processed in the order it is made.
- O2 : If request r by client c was made because a request r' from client c' caused it to, then r' is processed before r.
- We have seen this before :)

State machines and computer programs?

- How to implement SM's as programs?
- State machine implemented as procedures
- Client calls the procedure.

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- Chain Replication

Faults

- Byzantine Failures :
  - Arbitrary
  - Malicious behavior by faulty components.
  - Weakest possible failure assumption.
- Fail-stop failures: The good guys
  - Failure by stopping.

Failures...

Face it, join my team!

I killed a patient!

Byzantine

This is too much, I quit!!

Fail-stop

Tolerating faults

- *"t-fault tolerant"*
  - < t components become faulty
  - Makes failure assumption very explicit.
- Parameters such as MTBF.
  - More of a statistical measure
  - Not measuring degree of fault-tolerance.

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- 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
  - In the same order
  - Being deterministic, each will do the exact same thing
  - Produce the same output.

Really?

Like one House wasn't enough!

Headache,
fever

So now you
need a baby-sitter?

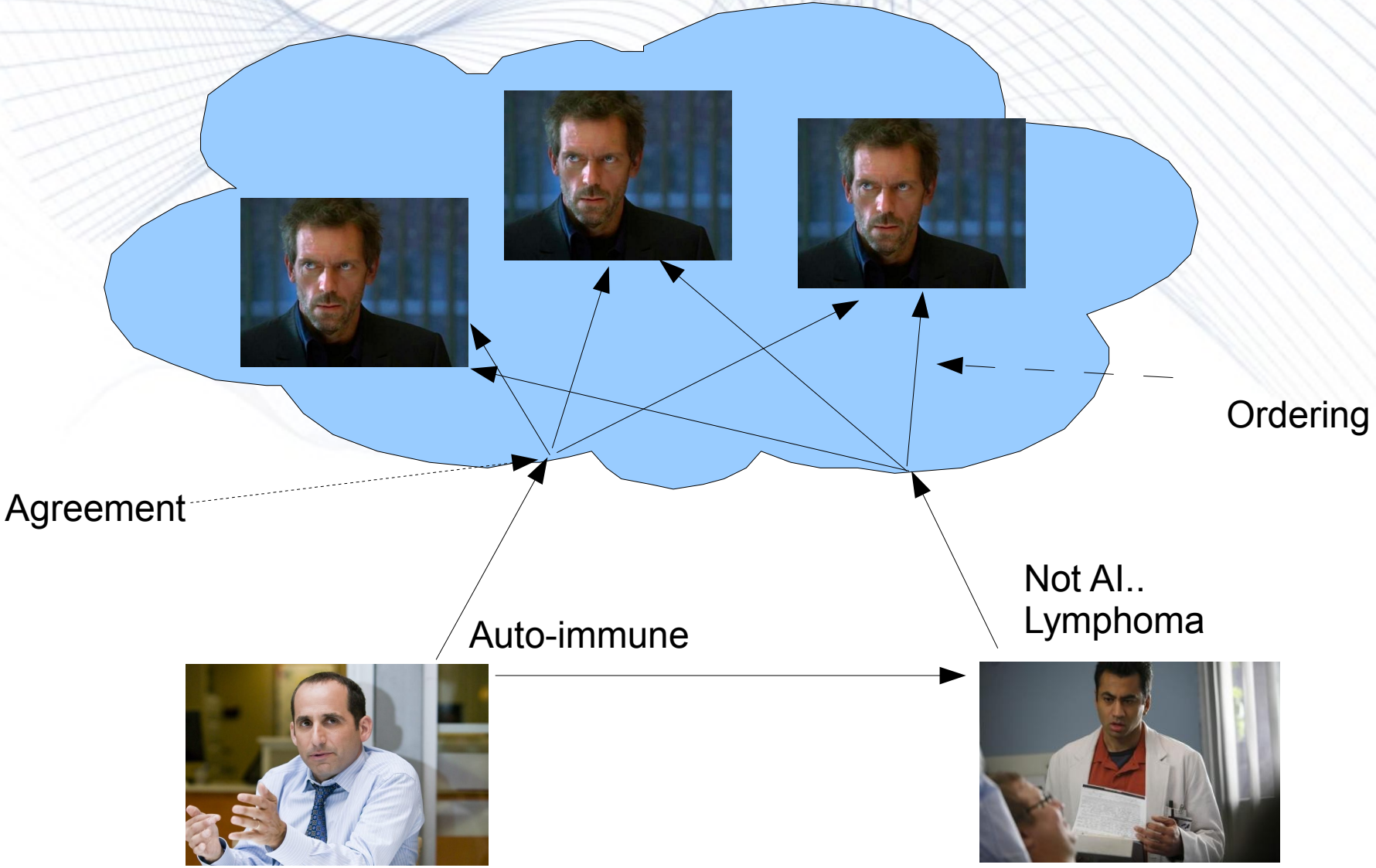
Headache,
fever

So now you
need a baby-sitter?

What all do we need?

- 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.

Agreement and Ordering



Outline

- State machines
- Faults
- State Machine Replication
  - Agreement
  - Ordering
- Failures Outside the state machines
- Other considerations
- 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
- Other considerations
- 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?

Ordering

- Can use synchronized real-time clocks.
- Max one request at every tick.
- If clocks synchronized within δ,
  - Message delay > δ
- Stability tests?

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').*
- Stability test?
- Potential Problems?
  - Could affect causality of requests
  - Client does not communicate until request is accepted.

How to generate uid's?

- Requirements:
  - UID1 and UID2 be satisfied
  - r != r' ⇒ *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....

- $cuid(sm_i,r) = max (\llcorner SEEN(i) \lrcorner , \llcorner ACCEPT(i) \lrcorner) + 1 + i/N.$
- $uid(r) = max (cuid(sm_i,r))$
- *Proof is simple.*

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- 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
  - Use properties of the system

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- 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 Combining function, P(t) - F(t)
- 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.

Outline

- State machines
- Faults
- State Machine Replication
- Failures Outside the state machines
- Other considerations
- Chain Replication

Papers!

- Chain Replication for supporting high throughput and availability

Robert van Renesse
FAST Search & Transfer ASA
Cornell University

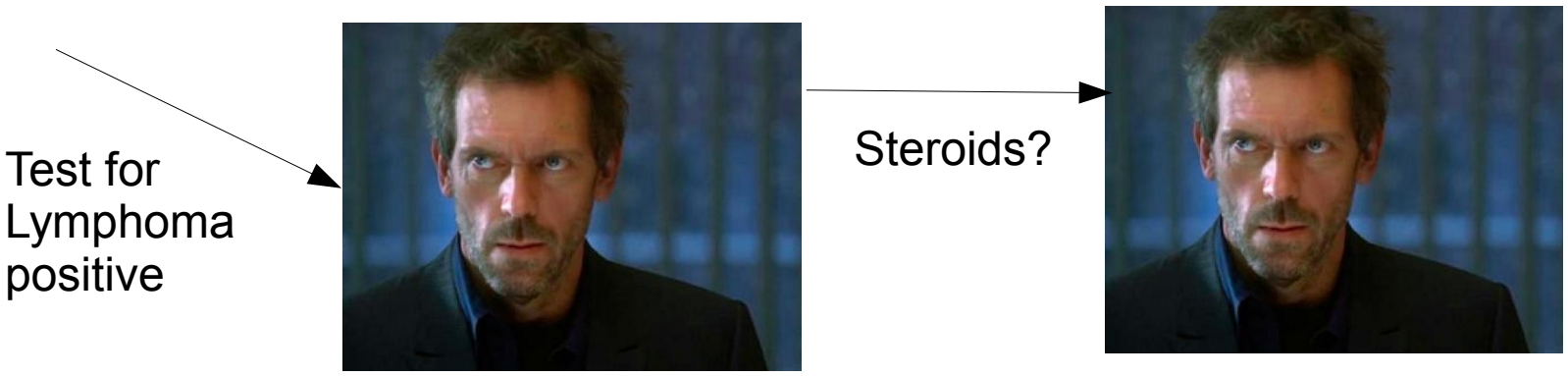
Fred B. Schneider
Cornell University

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.

Primary-Backup

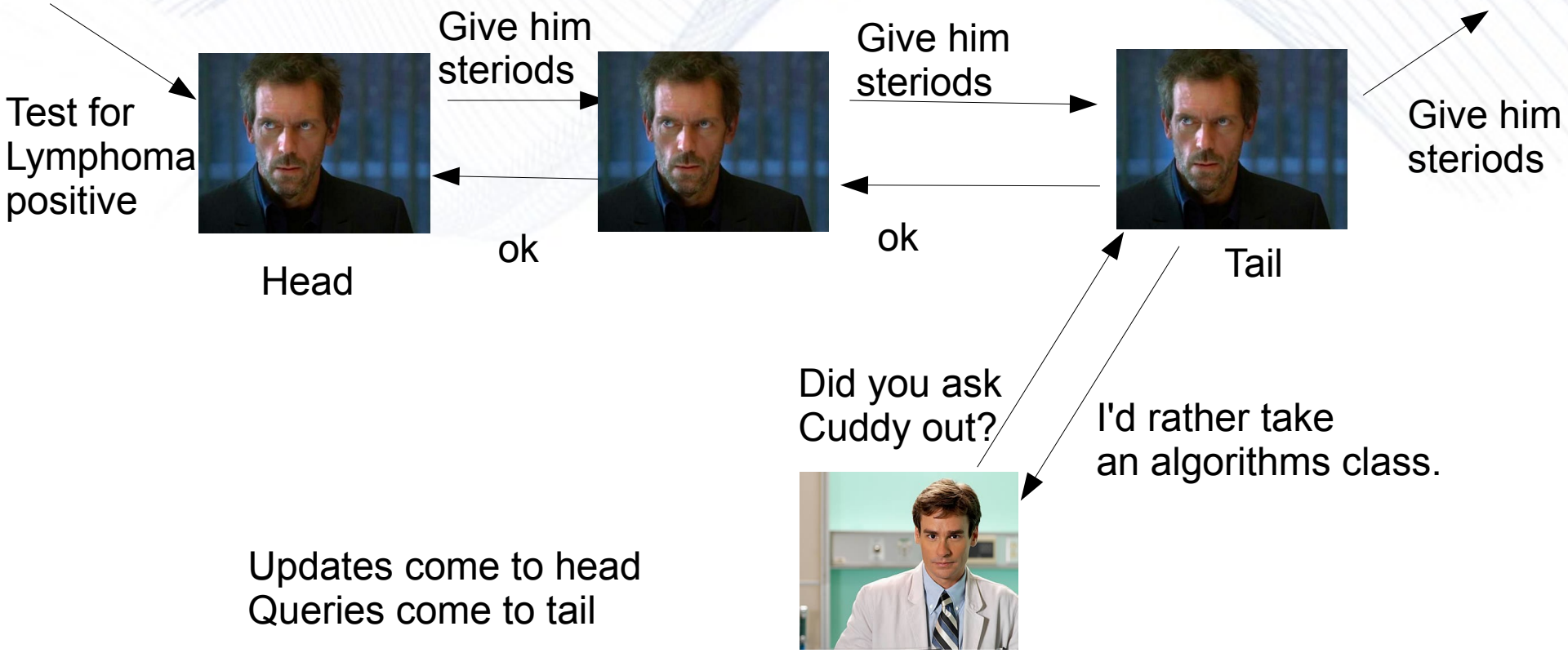
- Different from State Machine Replication?
- Serial version of State Machine Replication
- Only the primary does the processing
- Updates sent to the backups.



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.

Chain Replication



Updates come to head
Queries come to tail

Details....

- Client requests arrive at chain:
  - Either at head(updates) or tail(queries).
- Request processed by tail:
  - Execution of the request :modify state.
  - Send back ACK to predecessor.
  - Update state, send ACK to its predecessor...
- Request processed by head :
  - Updates are propagated down the chain.

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.

Head and Tail failure

- Head failed: Master removes head,makes successor head.
- Any updates not propagated are assumed lost, and the client may have to retransmit.
- Master removes the failed processor
- Makes the predecessor the new tail.
- Since the old tail's state is a subset of the current tail, it is easy to handle the failure of the tail.

Failure of other servers

- Master removes that server from the chain.
- Master informs its predecessor of its new successor.
- This server now transfers all un-ACKed requests to the successor.
- Successor sends ACKs based on which requests have been ACK-ed by its successor.

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.

Conclusions

- State Machine Replication probably would have less service interruption, but at a high cost, and usually more complex to implement.
- Primary-Backup easier to implement, reasonably high throughput and no need of consensus, so might be more efficient even in terms of network usage, but at the cost of lower availability.