Availability
(and Consistency)

Robbert van Renesse

Cornell University

Replication in the Fail-Stop Model

Robbert van Renesse
Cornell University

Replication History (as I understand it)

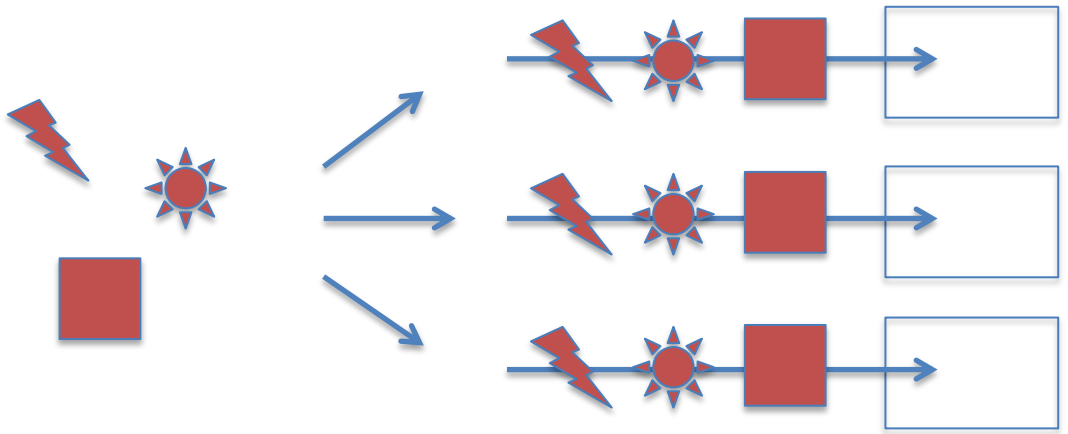
- Goes back until at least early seventies
  - A. Mullery. *The distributed control of multiple copies of data*. Technical Report RC 3642, IBM, Dec. 1971.
  - Driven by interest in fault tolerance and scalability.
- Highly influential RFC in mid seventies
  - P. R. Johnson and R. H. Thomas. *Maintenance of duplicate databases*. RFC 677, Jan. 1975.
  - Uses local timestamps, ties broken by processor id
  - Latest update wins (early example of "*eventual consistency*")
  - Conclusion: "*the probability of seemingly strange behavior can be made very small. However, the distributed nature of the system dictates that this probability can never be zero.*"

Early Solutions to *Strange Behavior*

- P. A. Alsberg and J. D. Day. *A principle for resilient sharing of distributed resources*. In Proc. of the 2nd IEEE Int. Conf. on Software Eng., ICSE '76.
  - now known as "*Primary-Backup*"
- L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM, 21(7):558–565, 1978.
  - introduces *logical timestamps* that capture causality
  - also introduces *replicated state machines*, although not fault-tolerant
- R. H. Thomas, *A solution to the concurrency control problem for multiple copy databases*. In Proc. of IEEE COMPCON 1978
  - introduces *majority voting*

What is *State Machine Replication*?

- A generic way to tolerate failures
- Objective: single copy behavior
- Simply start multiple *replicas* (copies) of a deterministic state machine, and keep them in sync by agreeing on the inputs and the order in which to apply them



System/Threat Models

- Assumptions about the environment in which a replication protocol runs
- Types of assumptions:
  - *Timing assumptions*
  - *Node failure assumptions*
  - *Communication assumptions*

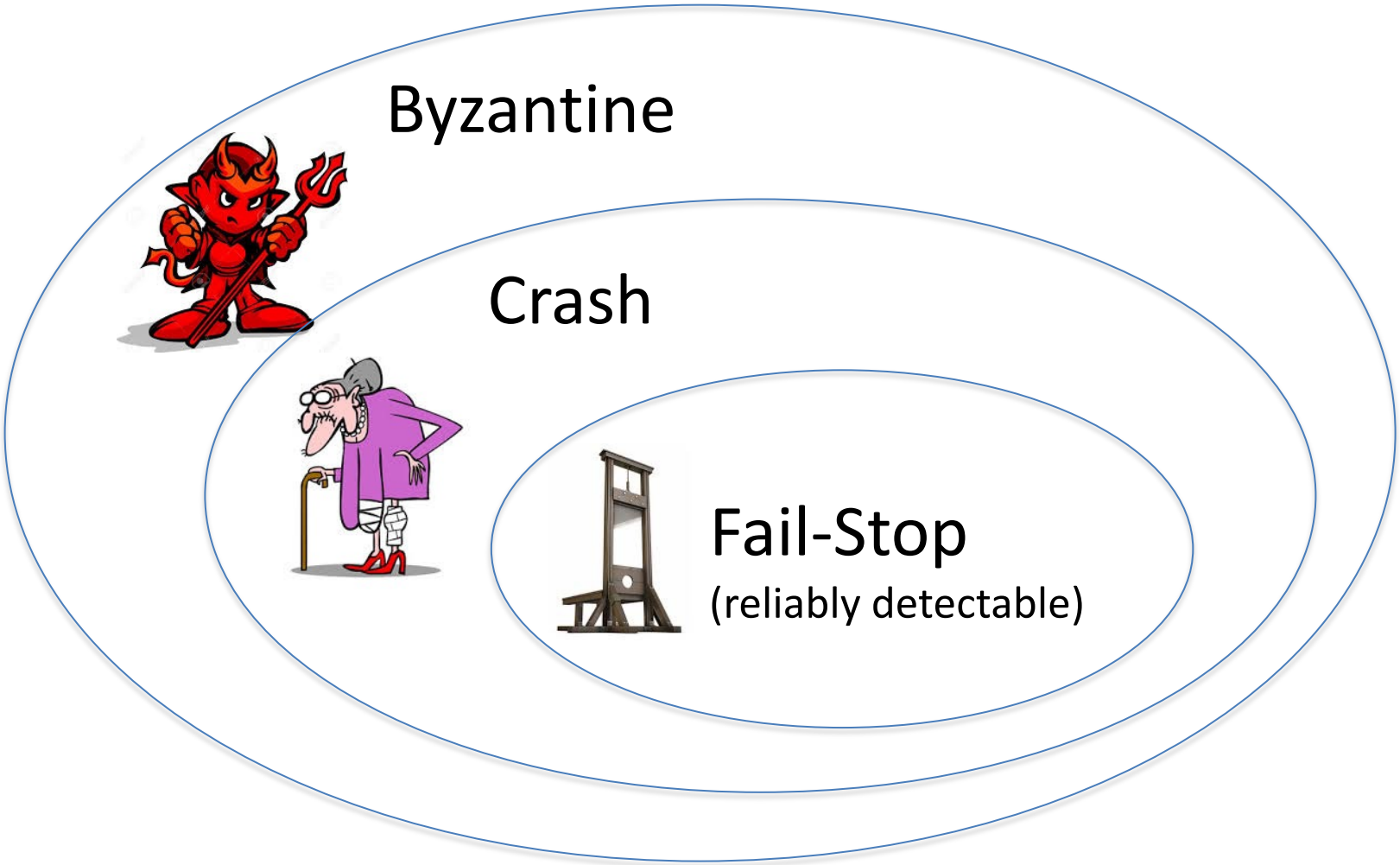
Communication Assumptions

- Loss, reordering, and duplication allowed
- Fair Links:
  - If processor *p*1 sends message *m* an infinite number of times to *p*2, *p*2 will deliver an infinite number of times
  - More practically: if *p*1 and *p*2 are correct, then a message sent by *p*1 to *p*2 is eventually delivered by *p*2
- Perfect Checksums:
  - If processor *p* delivers a message *m*, then some processor sent *m* to *p* (before that delivery)

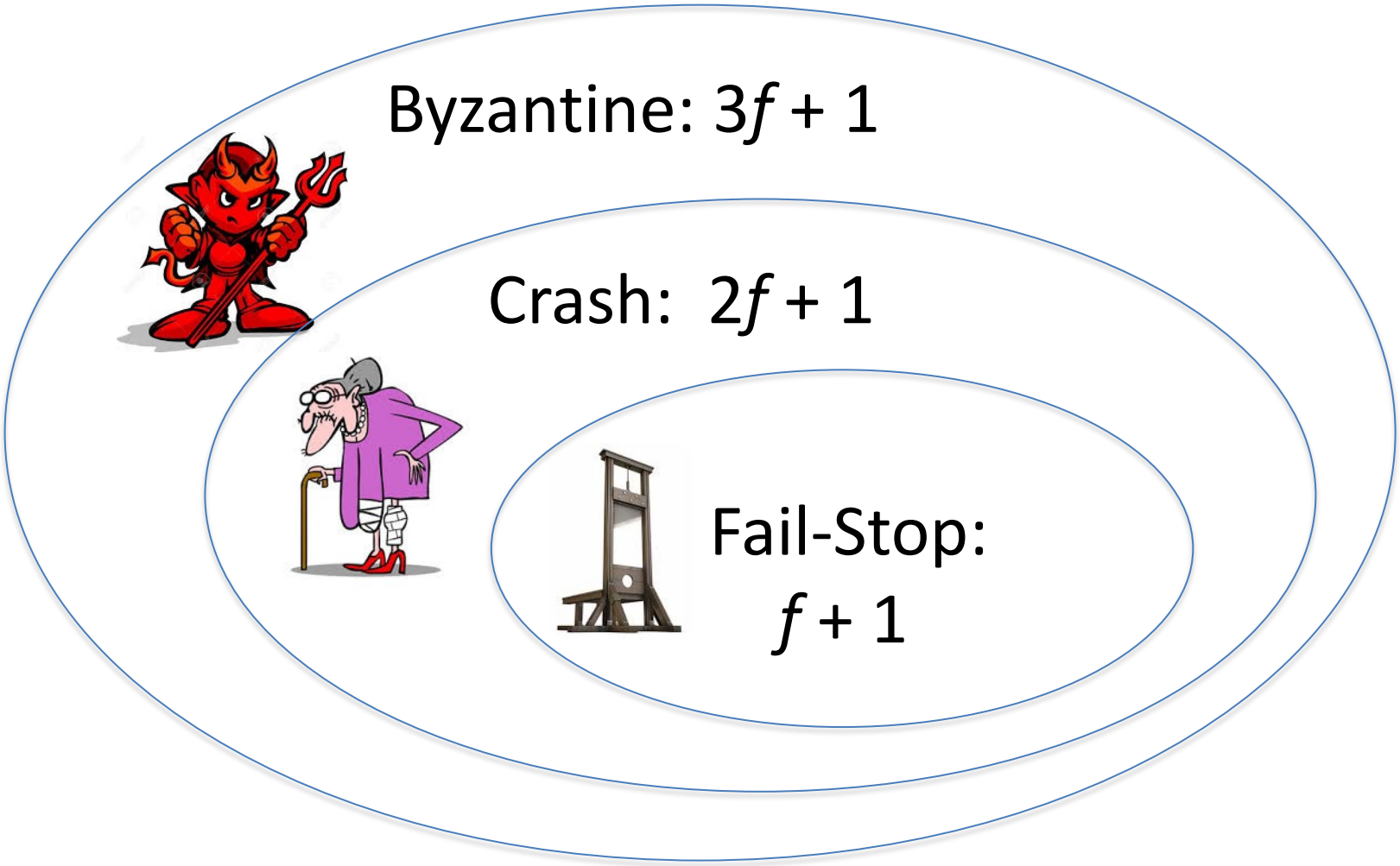
What is Asynchrony?

- No bounds on timing
  - no bounds on message latency
  - no bounds on how fast clocks run
    - but they do run monotonically increasing
  - no bounds on how skewed the clocks are
    - clocks on different machines show arbitrarily different times
  - no bounds on processing time
- Not to be confused with "*non-blocking*"
  - "asynchronous RPC" and "asynchronous system calls" are misnomers

Failure Type Hierarchy



Lower Bounds in Asynchronous Model
(#processors needed to tolerate *f* failures)



Our assumptions for today

- Fail-Stop
  - because replicas are expensive and Fail-Stop is a reasonable assumption in datacenters
- Asynchrony
  - because latency bounds would have to be very conservative and result in slow systems
- FIFO communication
  - because
    - it's cheap and easy to implement over fair links
    - fair links is a realistic assumption
    - checksums are close enough to perfect
    - simplifies life

Is Fail-Stop realistic? (in a datacenter)

- In absence of network partitions, failure detectors can be implemented that make mistakes with very low probability
- Redundancy in datacenter network topologies (fat tree / CLOS) make partitions extremely rare
- Even so, failure detectors occasionally suffer from false positives!
- But in a datacenter, false positives can be *eliminated*:
  - power-cycle suspected node; and/or
  - disable suspected node's network connections

An aside on disks

- Attaching a disk to a processor can make a processor more reliable
  - for example, with a disk, a processor may be able to mask temporary power failures
  - a power failure is just “the processor acting slow”, which means nothing in an asynchronous system
- *But disks can still fail*
- So *logically, there is no significant difference between a processor with or without a disk*

Existing Replication Protocols for the Fail-Stop Model

- *Primary-Backup* (Alsberg & Day, 1976)
- *Chain Replication* (Van Renesse & Schneider, 2004)

Both assume an external configuration service that reconfigures surviving replicas after a failure!

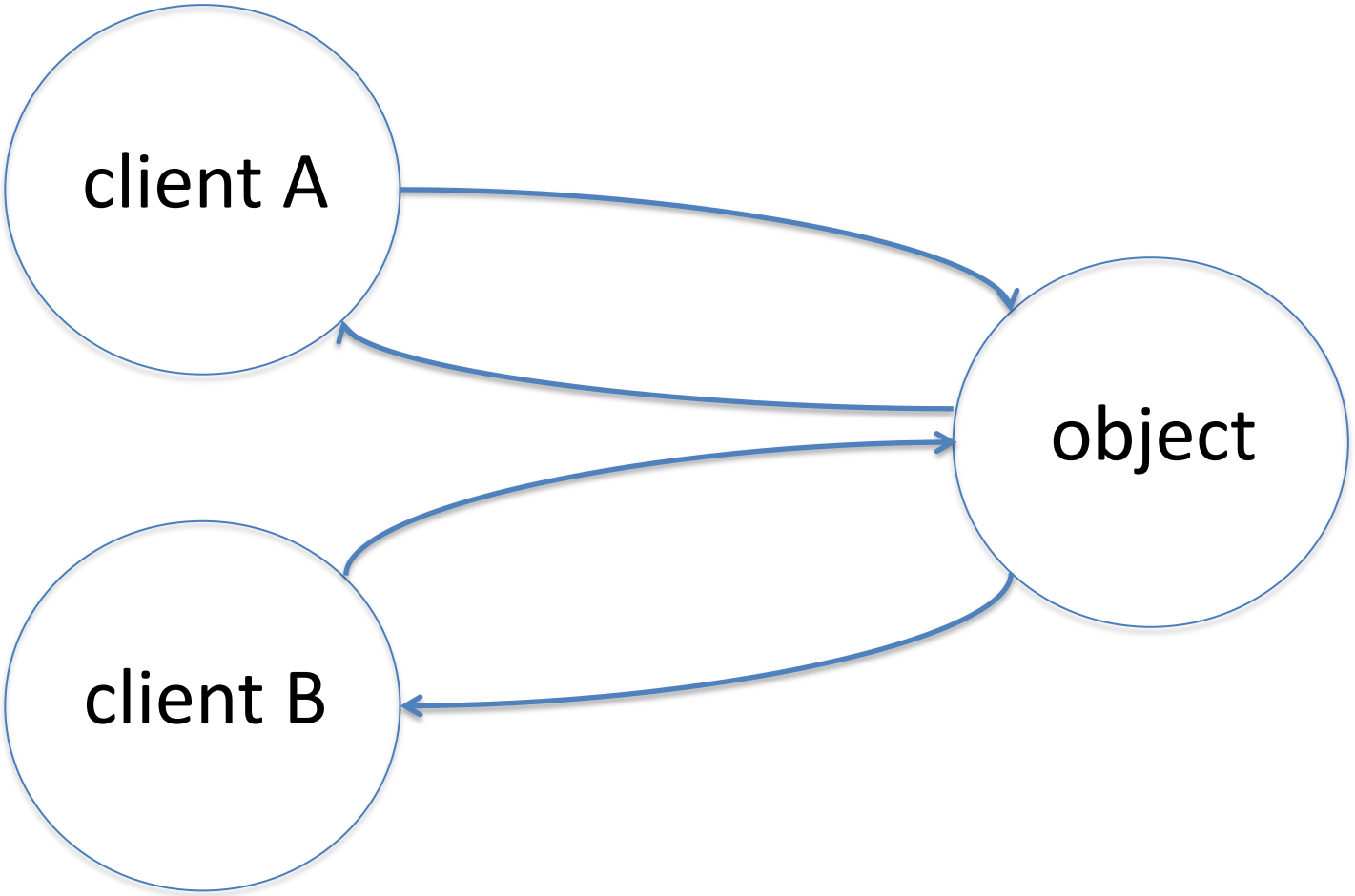
But how do you replicate the configuration service???

We'll show that you don't need such a service.

Specifying an Object to Replicate

- An object has
  - a *state*
  - a *query operation* that simply returns the state
  - one or more *update operations* that modify the state
  - we assume all operations execute *atomically*
- The current state of an object can be modeled as a pair consisting of
  - the *initial state* of the object
    - often implicit
  - the *history*: the list of update operations applied

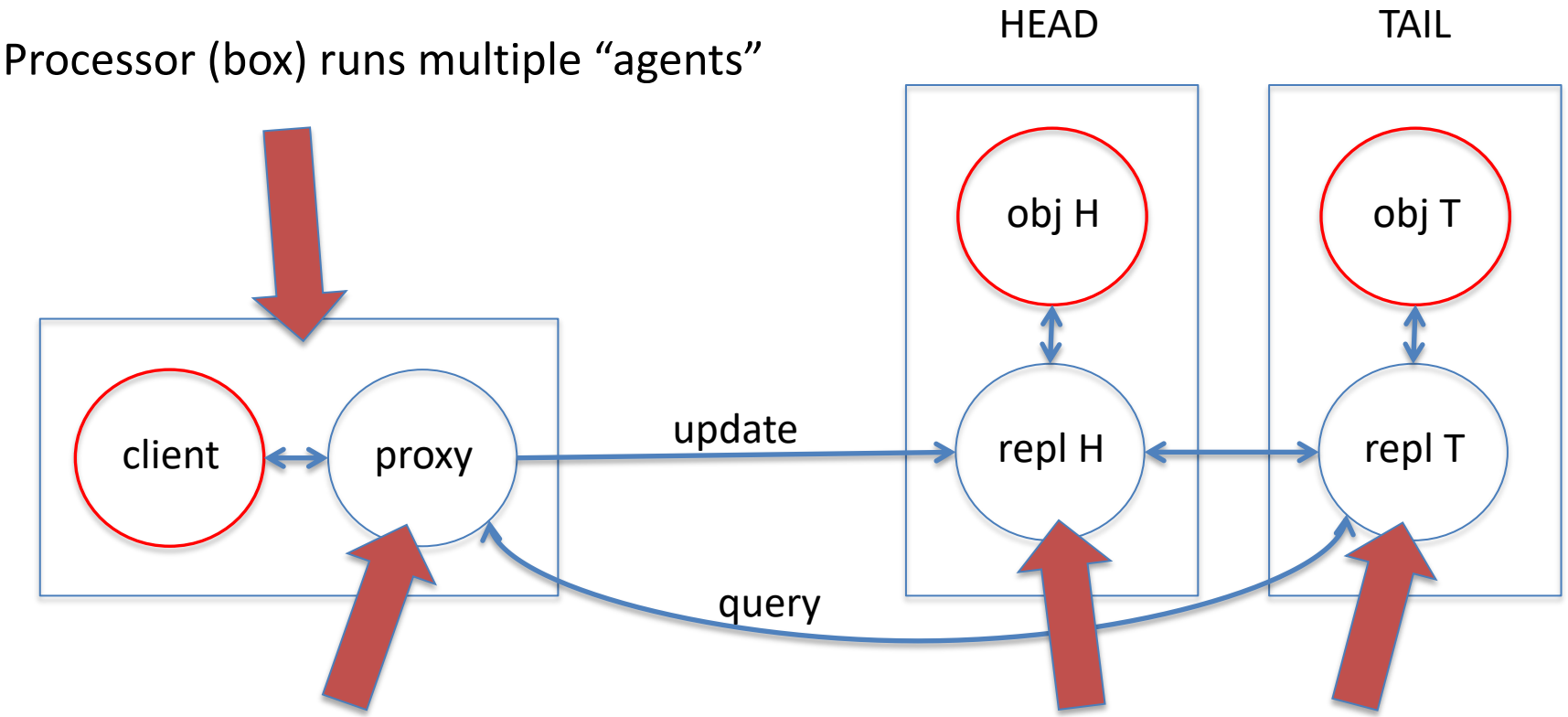
Client/Server Model + RPC



Tolerate 1 failure → 2 replicas

- A “head” and a “tail” replica
  - warning: don’t think of either as “primary”

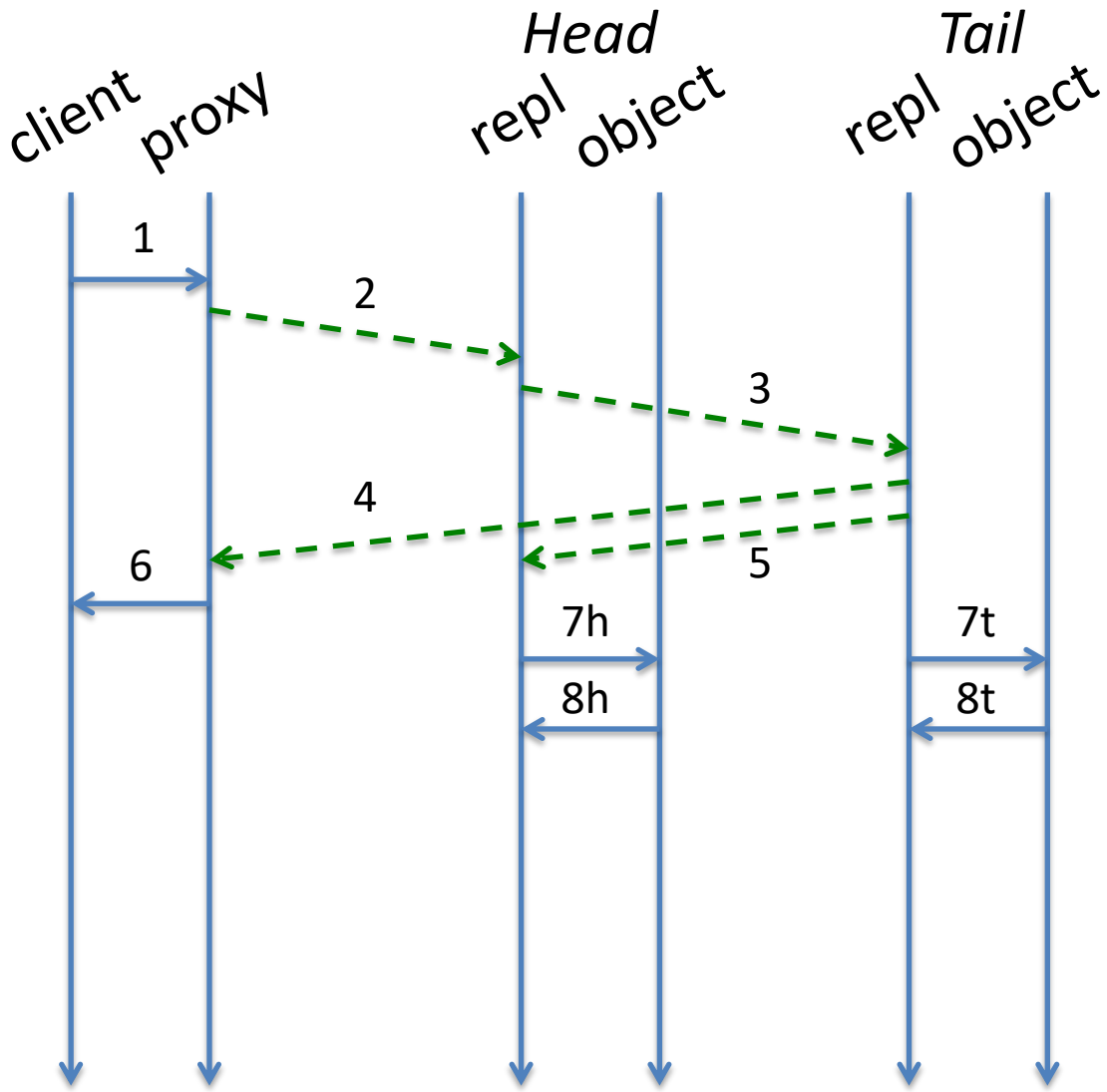
2 Replicas, “normal” operation



Proxy emulates object

Each replicator maintains a history (list of updates)

2 Replicas, normal update

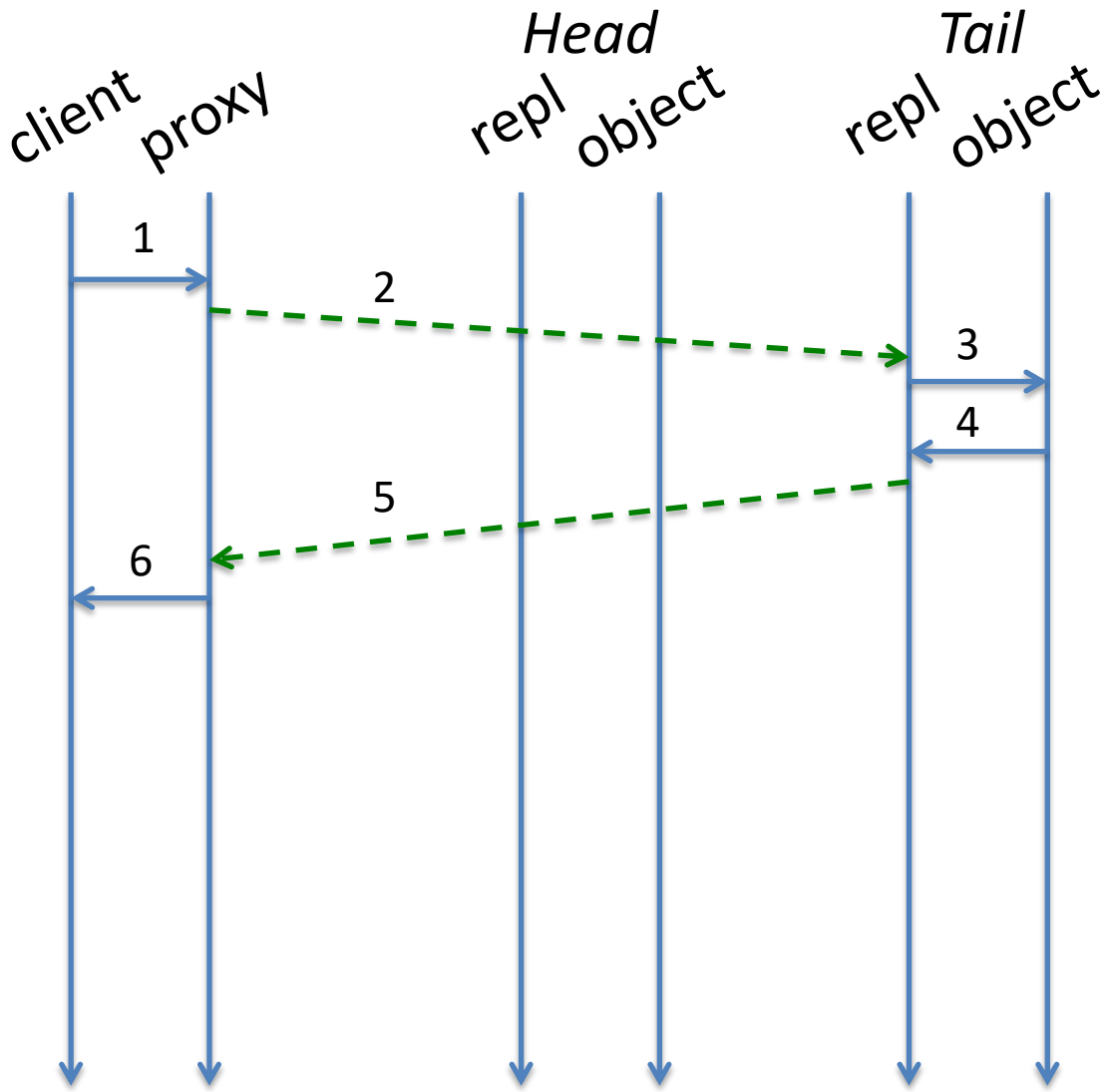


1. client sends update request
2. proxy forwards to head repl
3. head repl adds req to hist and forwards req to tail repl
4. tail repl adds req to hist and responds to proxy
5. tail repl sends ack to head repl
6. proxy responds to client
7. repls send request to objects
8. objects respond to repls

= over network

= within processor

2 Replicas, normal query



1. client sends query request
2. proxy forwards to tail repl
3. tail repl forwards to tail object
4. tail object responds to tail repl
5. tail repl responds to proxy
6. proxy responds to client

= over network

= within processor

Data held by replicators

- Each replicator maintains a
  - *speculative* history h
  - *stable* history $\underline{h}$ (= acknowledged by tail)
- Important invariants:
  - $\underline{h}_H \leq \underline{h}_T$: head's stable hist is prefix of tail's stable hist
  - $\underline{h}_T = h_T$: tail's stable hist equals tail's speculative hist
  - $h_T \leq h_H$: tail's spec hist is prefix of head's spec hist
  - Combined: $\underline{h}_H \leq \underline{h}_T = h_T \leq h_H$
  - Neither stable nor speculative history ever truncated
- Object replicas only see updates that are locally stable.
- Stable updates can be garbage collected
  - stable history is just an index into the speculative history

Refinement Mapping

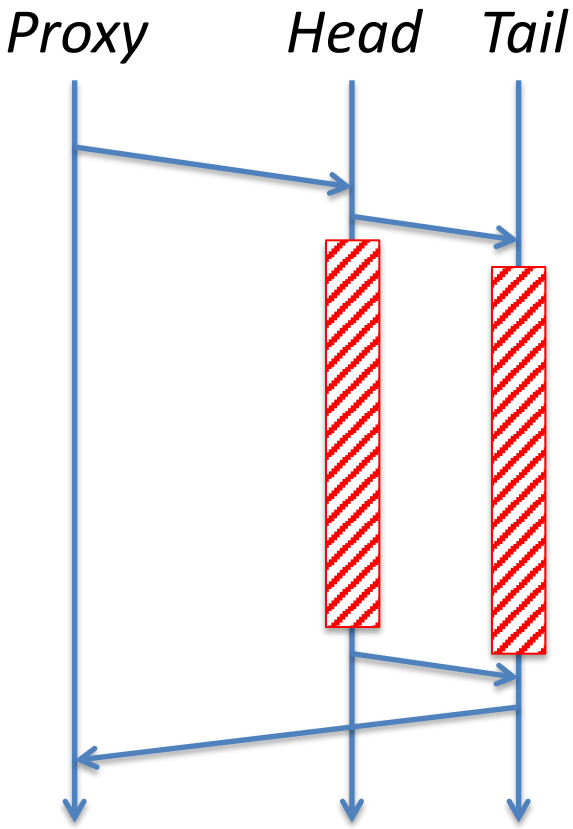
- The high level object's *current* state consists of
  - the initial state of the object
  - the tail replicator's speculative history
- The initial state of the high level object is the initial state of the tail replica's copy of the object
- A high level update "happens" (*linearization point*) when the update operation is added to the tail replicator's history.
- A high level query "happens" when the tail object replica sends the response that is ultimately forwarded to the client.
- All other low level transitions are "stutter transitions"

Bottom Line

- All the mechanism is basically just a complicated way to update the tail replica
  - leaving enough of a trail to be able to tolerate failures
- The head's history is "speculative" because it may never reach the tail

Disks Revisited

- Disks written one at a time
- OK under high load, as disk writes are pipelined
- But relatively high latency under low load
- Solution:
  1. head immediately forwards update to tail
  2. head and tail write disks (in parallel)
  3. head sends update completion message to tail
  4. tail responds to client on receipt



What about failures?

- Note that
  - a client only receives the response to an update *if it's stored by both replicators*
  - the history returned in a query is *stored by both the head and tail*
- *That's a good start*: all data that a client sees can survive one failure!
- And we're assuming there's at most one failure
- When there is a failure, the remaining processor becomes both the head and the tail

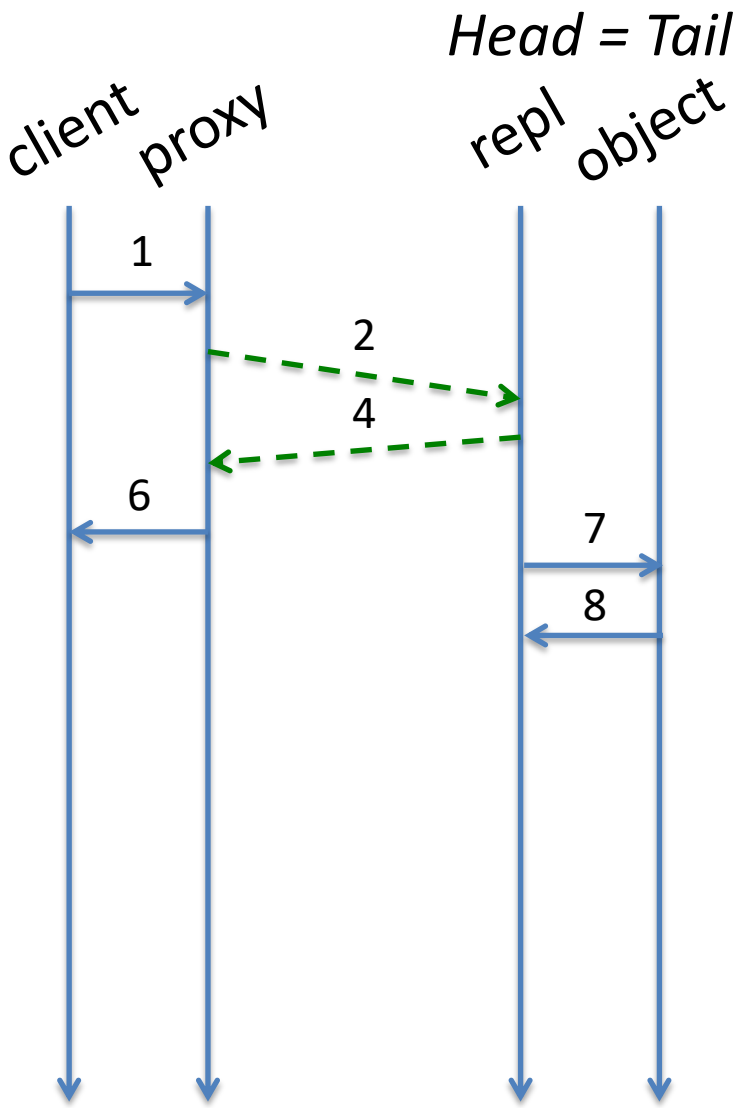
Case 1: head processor fails

- Tail becomes both head and tail
- Once failure detected, should ignore updates that were still en route from failed head
  - or, alternatively, re-order them
- Note: does not affect query processing
- The tail notifies the proxies about the failure
- A proxy may have outstanding updates that it has not received responses for
- The proxy retransmits those updates to the remaining replicator
- The replicator may receive updates it has already added to its history:
  - for each, the replicator should send a response to the proxy (and the proxy should filter out duplicate responses);
  - the replicator should otherwise ignore duplicate updates
- Proxy sends future updates directly to the remaining replicator

Case 2: tail processor fails

- Head becomes both head and tail
- By definition, its history becomes instantly stable
  - *crash of tail can correspond to "update" transitions in the refinement mapping!!*
  - *adding an update to the tail's speculative history and this crash event are the only transitions that map to updates*
- The replicator informs the proxies about the new configuration
- A proxy may have outstanding updates *and* queries that it has not received responses for
- The proxy retransmits those operations to the remaining replicator
- Replicator may receive updates it has already added to its history
- Proxy filters out duplicate responses to both updates and queries
- Proxy sends future queries directly to the remaining replicator

1 Replica, “normal” update



1. client sends request
2. proxy forwards to head repl
3. ~~head repl adds req to hist and forwards req to tail repl~~
4. tail repl adds req to hist and responds to proxy
5. ~~tail repl sends ack to head repl~~
6. proxy responds to client
7. repl sends request to object
8. object responds to repl

= over network

= within processor

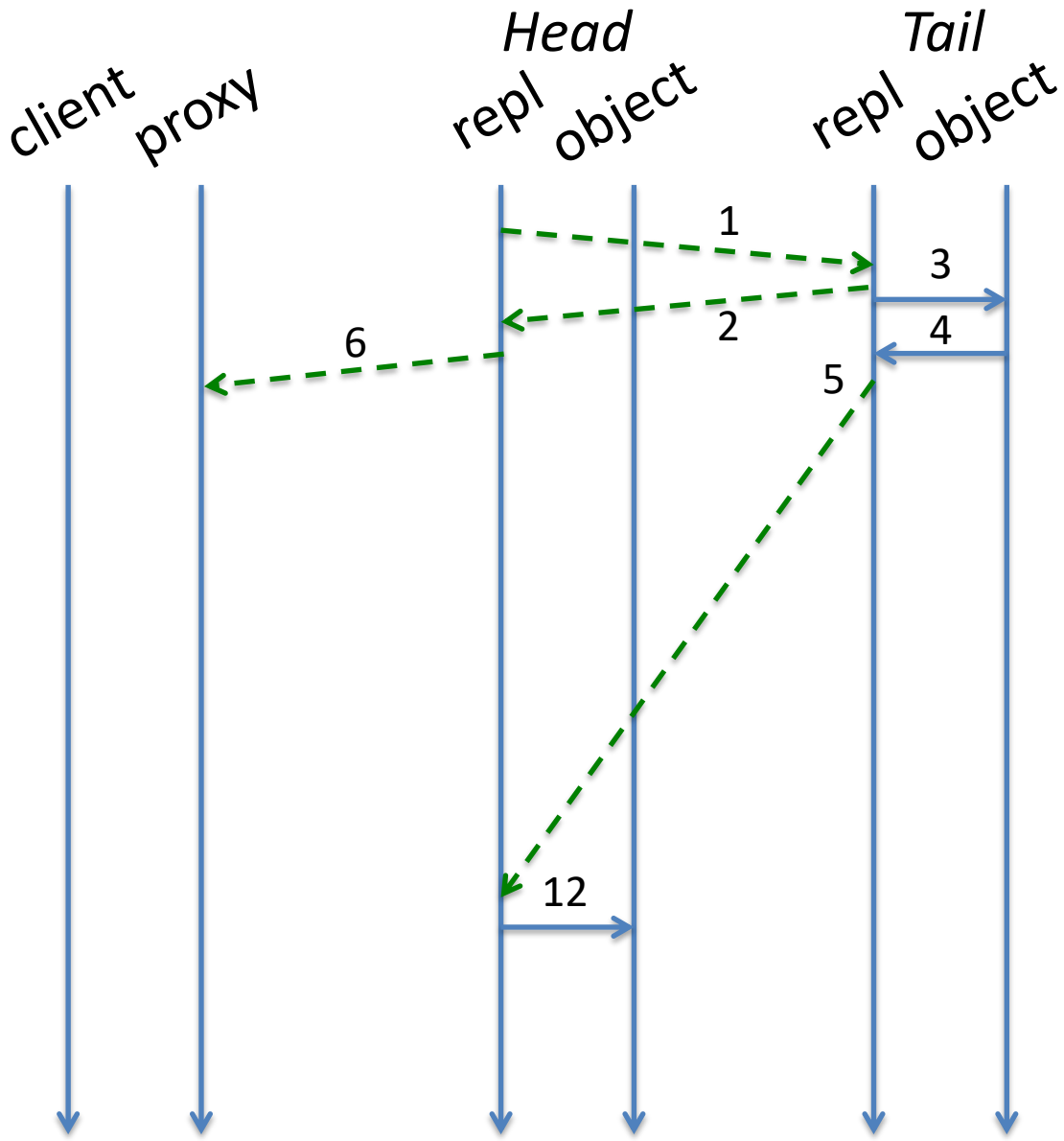
End-to-End Considerations

- The proxies add, to each update, a unique identifier consisting of the client identifier and a sequence number
  - to prevent duplicate updates on histories, and
  - so that the tail replicator knows where to send the response.
- Requires that replicators keep a client → sequence number mapping
  - also replicated, just like the object
- The uid is also included by the tail replicator in the response to the proxy
  - to filter out duplicate update responses.

Adding a new replica (in case there is only one left)

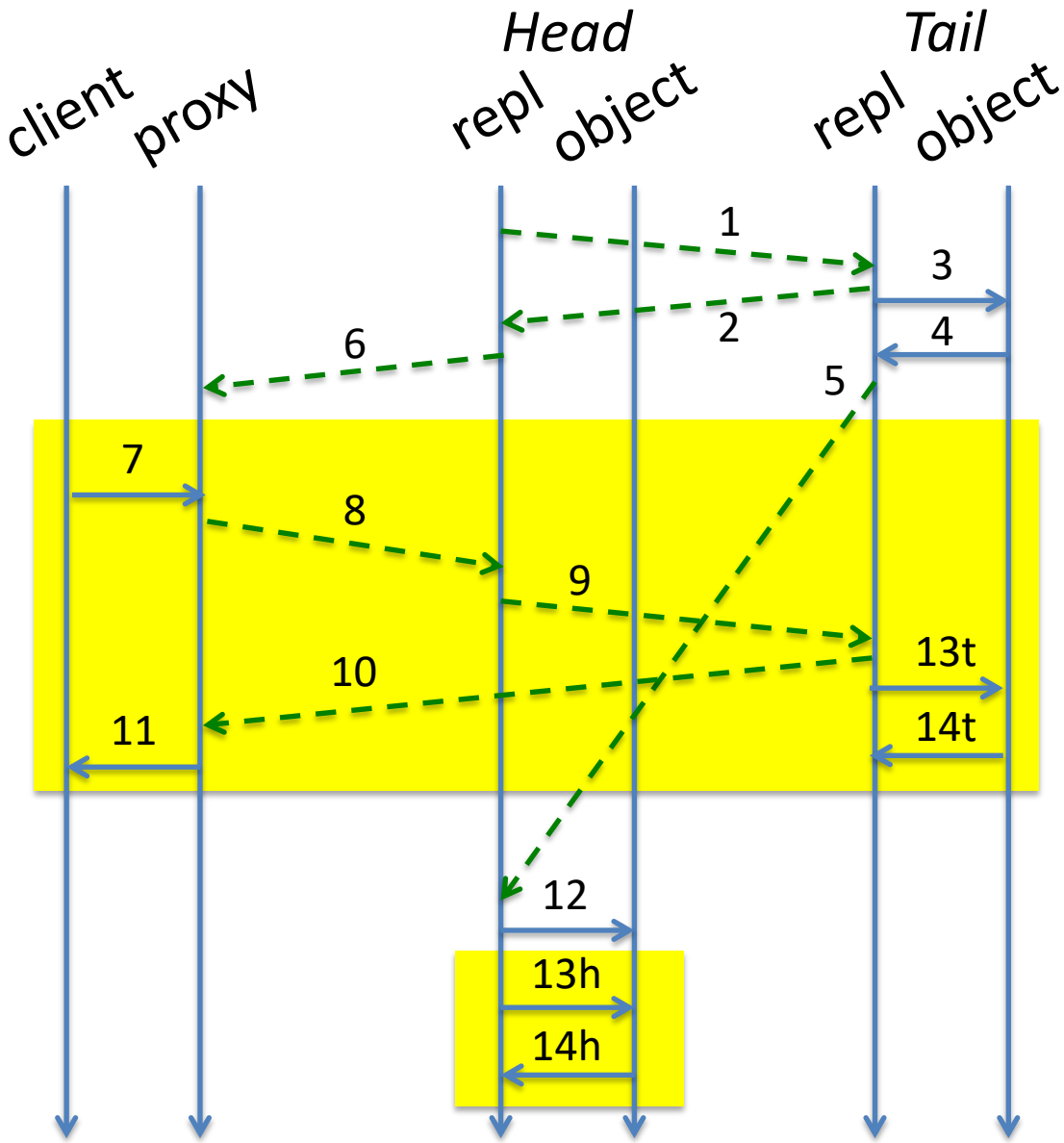
- New replica can become the head or the tail
  - Making it the head is easier because of the refinement mapping
- Let X be new replicator, Y existing replicator
- Steps:
  1. X sends “join” to Y
  2. Y makes sure it is both head and tail currently. Then Y sends *join response* to X: #stable updates and client → sequence number map
  3. Y takes a snapshot of the object’s state and start streaming the snapshot to X (in background)
  4. On receipt of join response, X informs proxies of new head
  5. X starts accepting update requests and forwarding them to the tail
  6. X cannot deliver stable updates to its local object until it has received and delivered the entire snapshot
  7. Adding replica is complete after delivery of snapshot. Until then, no failures can be tolerated
  8. All these actions are stutters with respect to refinement mapping, as tail doesn’t change

Join Protocol



1. new repl sends join request
2. old repl responds
3. old repl requests snapshot
4. object responds with snapshot
5. tail repl streams snapshot to new head replicator
6. head replicator informs proxies
7.
8.
9.
10.
11.
12. head repl delivers snapshot

Join Protocol



1. new repl sends join request
2. old repl responds
3. old repl requests snapshot
4. object responds with snapshot
5. tail repl streams snapshot to new head replicator
6. head replicator informs proxies
7. client sends update request
8. proxy forwards to head repl
9. head repl forwards to tail repl
10. tail repl responds to client
11. proxy responds to client
12. head repl delivers snapshot
13. repls delivers update
14. objects respond

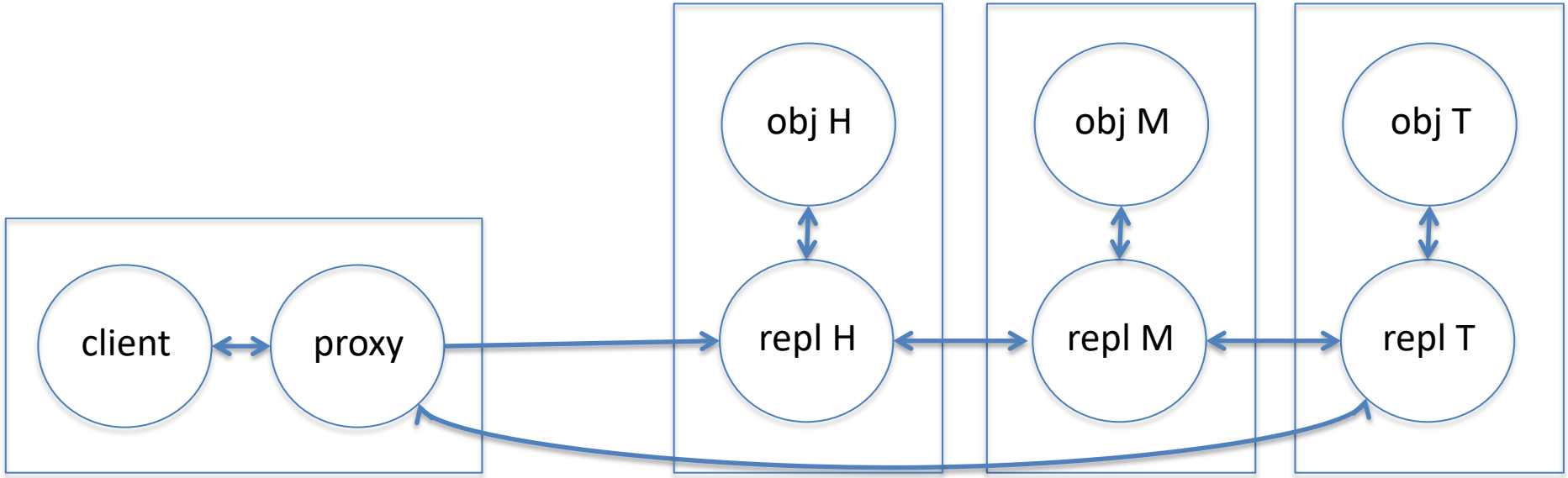
Join Protocol, salient properties

- no external configuration service needed
- if two processors try to join at the same time, only one will be successful
  - only a replicator that is both head and tail responds
- state transfer is entirely in background
- if head crashes during state transfer, tail aborts state transfer and continues by itself as before
- after state transfer, tail failure can be tolerated

Generalizing to >2 Replicas

- Basic intuition: we can replicate the head, so we end up with a head-head and a head-tail and a total of three replicas
  - could've done the same thing to the tail
- Doing so gives rise to a chain of replicas

Chain Replication



invariants:

$\underline{h}_T = h_T$: tail's stable hist equals tail's speculative hist

$\underline{h}_H \leq \underline{h}_T$: head's stable hist is prefix of tail's stable hist

$\underline{h}_M \leq \underline{h}_T$: same for middle replica

$h_T \leq h_M \leq h_H$: spec hists are (reversely ordered)

(it's not necessary that $\underline{h}_H \leq \underline{h}_M$ holds)

Failure Recovery when N > 2

- In voting protocols such as Paxos, operation continues with up to *f* failures
  - in particular Paxos can decide on new configurations
- In Chain Replication, ordering comes to a halt even if there is just one failure
- When N > 2, the remaining replicas have to coordinate on recovery
  - they may not detect failures in the same order
  - multiple nodes may be trying to join as well

Reconfiguration Operations

- Special operations that control reconfiguration:
  - addReplica(processor id)
    - used by a processor to join (and become head)
  - removeReplica(processor id)
    - used to remove a processor if the head is still alive
  - resumeAsHead(processor id)
    - processor notifies that it has become the head again

Speculative Configuration

- Each replica maintains a "speculative configuration" based on the configuration operations it has in its speculative history
- A configuration command becomes stable when it reaches the tail

Removing Failed Replicas

- When a replica detects the failure of another replica *p* (that is not the head), it sends removeReplica(*p*) to the head (according to its speculative configuration)
- removeReplica(*p*) is routed like any other operation, but updating the speculative configuration of each replica that receives it
- Updating the speculative configuration of a replica *q* may cause its successor in the chain to change
- Two cases:
  - *q* becomes the tail: *q* notifies the proxies that there is a new tail to send queries to
  - *q* gets a new successor: *q* retransmits its (unstable) speculative history to its new successor

Removing Failed Head Replica

- If the head fails, there is no replica to send removeReplica(head) to
- Instead, when a replica *p* detects the failure of all its predecessors, it adds resumeAsHead(*p*) to its speculative history to indicate that it has (once again) become head

Notifying Proxies

- Proxies should be notified when there is a new head (either due to joining or to the old head failing) or a new tail (due to the tail failing)
- Proxies are only notified about stable configuration updates
- Stable configurations are numbered so proxies can distinguish the most recent configuration

Speculative History Revisited

- There are three types of operations in the speculative history
  1. object update operations
  2. reconfiguration operations
  3. add/remove client operations
- These give rise to three speculative “states” that a replica maintains:
  - speculative object state
  - speculative configuration
  - speculative client registry
    - maps client ids to sequence numbers

Conclusion

- Chain Replication a cheap and credible replication scheme in datacenters (or any place where Fail-Stop is a reasonable assumptions)
  - deployed by Microsoft Azure Blob store and a bunch of other commercial and open-source storage services
- Fail-stop protocols can reconfigure themselves and recover from “total failure”