CSE535: Asynchronous Systems
Fall 2014

Chain Replication

Scott D. Stoller

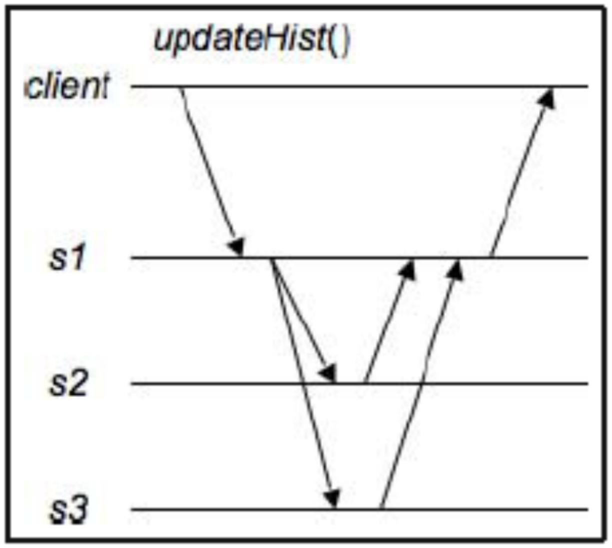
Stony Brook University

References

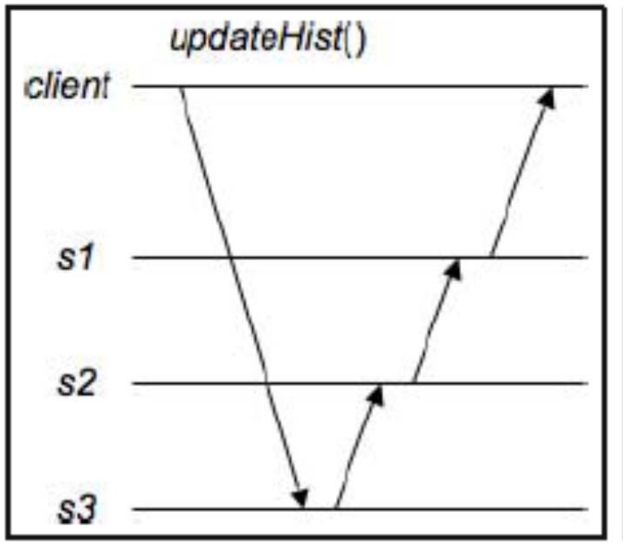
- Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91-104. USENIX Association, 2004.
  - http://www.cs.cornell.edu/fbs/publications/ChainReplicOSDI.pdf
- Robbert van Renesse and Rachid Guerraoui. Replication Techniques for Availability. In Replication: Theory and Practise, Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper, editors, volume 5959 of Lecture Notes in Computer Science, pages 19-40. Springer-Verlag, 2010.
  - http://www.cs.cornell.edu/Courses/cs5414/2012fa/publications/vRG10.pdf

Introduction

- Specification and design of a fault-tolerant distributed object storage service with relatively strong consistency guarantees and good performance.
- Fault-tolerance is achieved through chain replication, a variation of primary-backup, with chain-like instead of star-like communication pattern.



Primary-Backup



Chain Replication

Metrics

- Performance
  - Throughput: requests/second
  - Latency: time from sending request to receiving reply
- Availability
  - Time to reconfigure and resume service after a failure.
  - Number of requests dropped as a result of a failure
- For the intended applications, throughput is more important than latency.

API

- Distinguish two kinds of operations: queries and updates.
- Updates can return values, so queries are a special case.
- Distinguish queries because they can be implemented more efficiently.
- query(objID, opts)
    - Think of “opts” as “args”.
- update(objID, newVal, opts)
    - Effect: new state = F(current value, newVal, opts)
    - Calling the argument “newVal” is somewhat confusing.
    - The operation F may be non-deterministic.
        - Example of a non-deterministic operation on a collection: return any element satisfying …

Specification

- Specification of the service (not individual servers), expressed as a state machine.
- The specification implies these consistency guarantees:
  - Operations on an object (appear to) occur in some sequential order, called the serialization order.
  - A completed update is reflected in all subsequent operations.
- State Variables:
  - Hist_o: sequence of updates to object o
  - Pending_o: set of pending requests for object o
- Transition T1: Client issues request r for object o.

  $$\text{Pending}_o = \text{Pending}_o \cup \{r\}$$
- T1 is non-deterministic: r and o are unspecified.

Specification

- Transition T2: Request r in Pending_o is dropped (ignored).

  $\text{Pending}_o = \text{Pending}_o - \{r\}$
- T2 may occur as a result of communication failures or server failures (i.e., crashes).
- Server failures could be masked, but…
  - The algorithm is more complicated and expensive.
  - It would provide little benefit, because clients would still need to deal with communication failures.
- What should a client do if it does not a receive a timely reply to a request?

Specification

- Typical application behavior if timely reply not received for a query: retransmit it.
- Typical application behavior if timely reply not received for an update:
  - If update is idempotent, retransmit it.
  - If update is not idempotent, one approach is to use an (application-specific) query to determine whether the update was processed, and if not, retransmit it.
  - If the query is submitted too soon, it could be processed by tail while update is propagating down chain.
  - An alternative that avoids this is for the client to submit a request for a special update that performs that query and then performs the update if the query indicates it was not already processed.

Specification

- Transition T3: Request r in $Pending_o$ is processed.

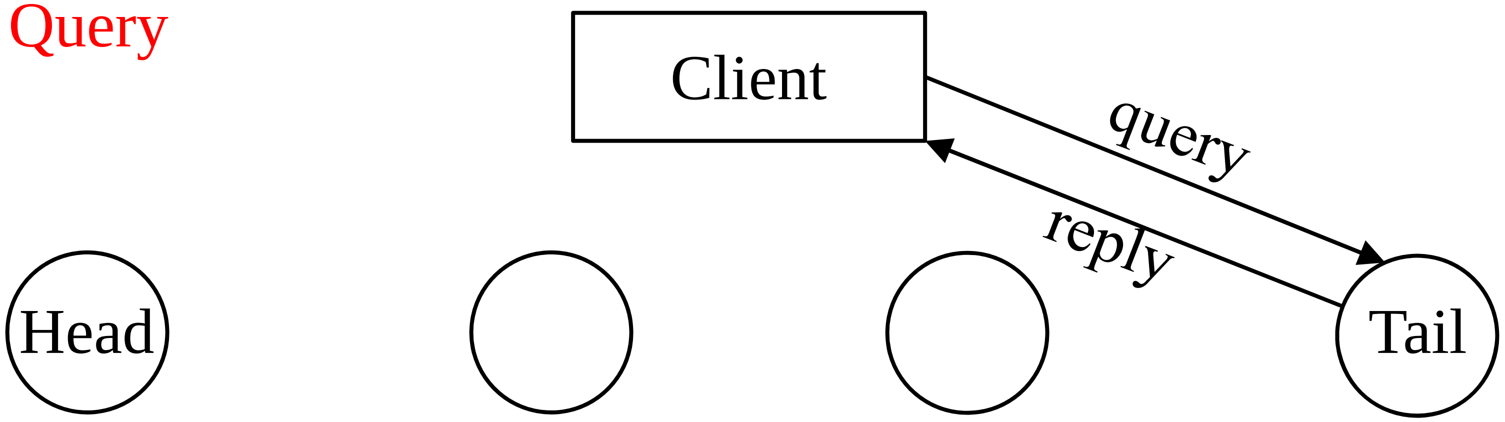
```
$Pending_o = Pending_o - \{r\}$
if (r is query(o,opts))
    compute reply from $Hist_o$ and opts
else if (r is update(o, newVal, opts))
    $Hist_o = Hist_o \cdot r$
    compute reply from $Hist_o$, newVal, and opts
```

Assumptions and Requirements

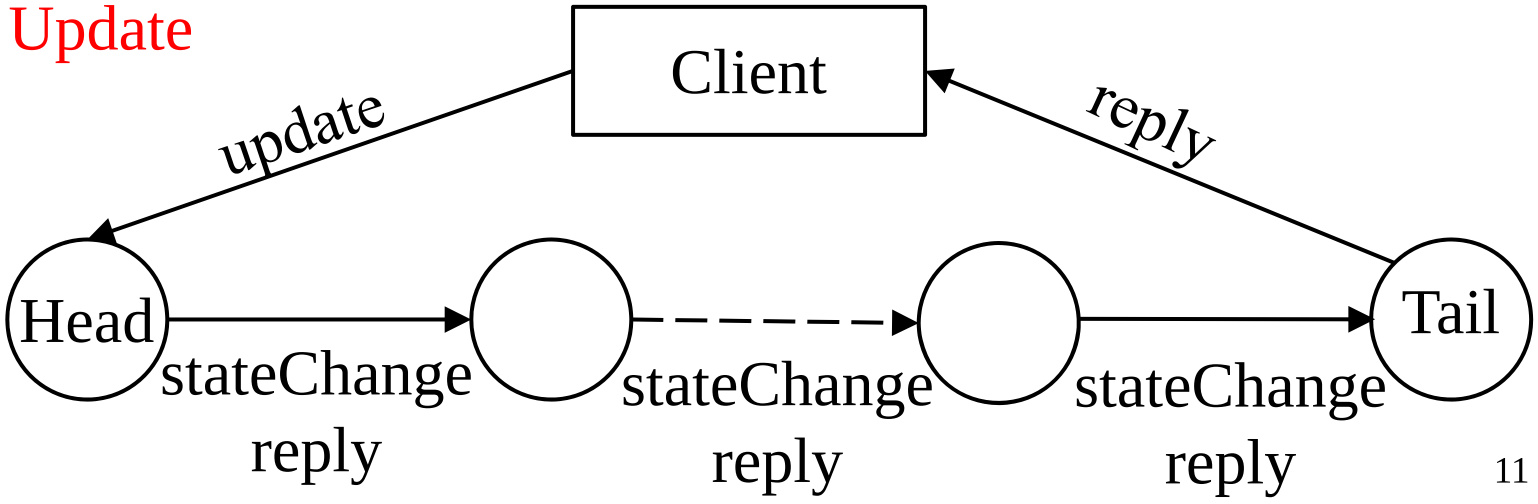
- Assumptions
  - Servers are fail-stop.
    - This encapsulates necessary synchrony assumptions.
  - Communication between servers is reliable
  - Communication between clients and servers is unreliable.
- Requirements
  - The service satisfies the specification if at least one server is alive.
  - New servers can be added.

Overview of Algorithm

- Query



- Update



Proof of Correctness Without Failures

- Informally, in the absence of failures, consistency is guaranteed because the tail replies to all requests, so the tail defines the serialization order.
- For a more detailed correctness proof, define a refinement mapping A.
- Notation: $Hist_o^S$ = the value of $Hist_o$ stored by server S
- State of state machine in spec. = A(implementation state)
- A(implemState).$Hist_o$ = $Hist_o^T$
- A(implemState).$Pending_o$ = requests received by any server in the chain and not yet processed by the tail.
  - The service isn't responsible for requests that might get dropped before they reach the chain, so those requests can be omitted from $Pending_o$.

Proof of Correctness Without Failures

- Show: Each transition of the implementation is correct because there is a corresponding transition of the specification state machine.
- Formally, if $s_1 \rightarrow_{impl} s_2$, then $A(s_1) \rightarrow_{spec} A(s_2)$.
- Two kinds of implementation transitions change A(s).
  - Other implementation transitions are mapped to a stutter transition of A, i.e., to a transition $s \rightarrow_{spec} s$.
- 1. A request r for object o is received by a server S (head or tail). The implementation transition adds r to $s_2.Hist_o^S$. By definition of A, $A(s_2).Pending_o = A(s_1).Pending_o \cup \{r\}$. r must have been sent by a client, so the specification state machine can take a T1 transition that adds r to $Pending_o$. This implies $A(s_1) \rightarrow_{spec} A(s_2)$.

Proof of Correctness Without Failures

- 2. A request r for object o is processed by the tail. The implementation transition appends r to Hist_o^T and sends reply.
  - Note: If r is an update, the state change and reply were originally computed by H using Hist_o^H. This is OK because the current value of Hist_o^T is the value of Hist_o^H when they were computed.

By definition of the refinement mapping A, r is removed from Pending_o, i.e., $A(s_2).\text{Pending}_o = A(s_1).\text{Pending}_o - \{r\}$, and if r is an update, then r is appended to Hist_o, i.e., $A(s_2).\text{Hist}_o = A(s_1).\text{Hist}_o \cdot r$. These are exactly the effects of a T3 transition of the specification state machine, so $A(s_1) \rightarrow_{\text{spec}} A(s_2)$.

Coping With Server Failures

- Reconfigure the chain to eliminate failed (crashed) servers.
- Master process M
  - Detects server failures
  - Informs neighbors of the failed server of their new predecessor or successor
  - Informs clients of the new head or tail, if the head or tail failed
- Assume the master never fails.
  - In the implementation, the master is a fault-tolerant service provided by a group of processes coordinated using Lamport's Paxos algorithm. We ignore this level of detail.

Coping With Server Failures

- Notation: S^+ = successor of S in chain, S^- = predecessor of S in chain.
- Handling of failures is guided by desire to maintain the…
- Update Propagation Invariant (UPI): $i \leq j \Rightarrow Hist_o^j \preccurlyeq Hist_o^i$
    - $i \leq j$ means i precedes j in the chain
    - $h_1 \preccurlyeq h_2$ means h_1 is a prefix of h_2
- It is easy to see that UPI holds in the absence of failures.

Failure of Head

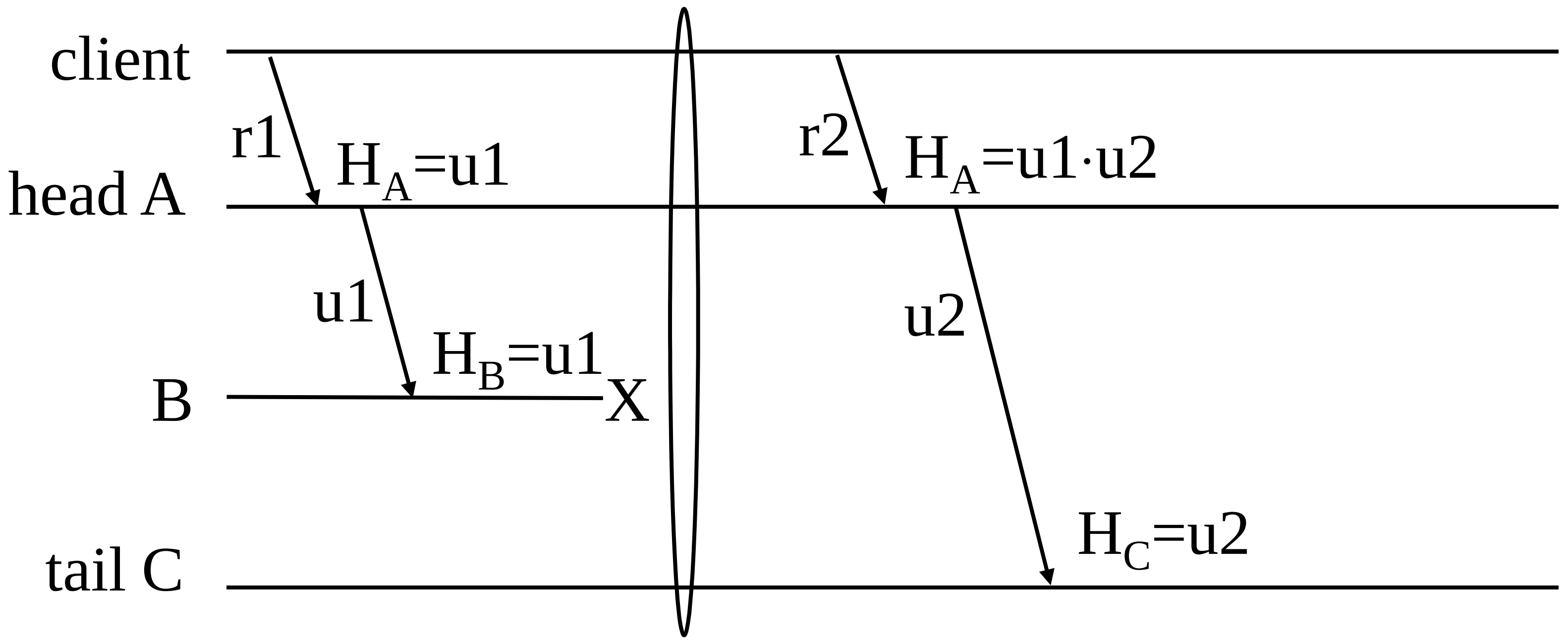
- Upon detecting failure of head H, the master removes H from chain by telling H^+ it is the head and telling clients that H^+ is the head.
- Let s_1 and s_2 denote the implementation state before and after these changes, respectively. Show: $A(s_1) \rightarrow_{spec} A(s_2)$.
- By definition of A, $A(s_2).Hist_o = A(s_1).Hist_o$, because tail does not change.
- By definition of A, $A(s_2).Pending_o = A(s_1).Pending_o$ – {requests received by H and not by other servers in chain}.
- T2 transitions in the specification state machine have this effect, so $A(s_1) \rightarrow_{spec} A(s_2)$.
- UPI is not affected (hence still holds), because all $Hist_o^i$ are unchanged.

Failure of Tail

- Upon detecting failure of tail T, the master removes T from the chain by telling T^- that it is the tail and telling clients that T^- is the tail.
- Let s_1 and s_2 denote the implementation state before and after these changes, respectively. Show: $A(s_1) \rightarrow_{spec} A(s_2)$.
- By UPI, $Hist_o^T \preccurlyeq Hist_o^{T-}$. Let $X = \{Hist_o^{T-} - Hist_o^T\}$ = requests processed by T^- and not by T.
- By definition of A, $A(s_2).Pending_o = A(s_1).Pending_o - X$, and $A(s_2).Hist_o = A(s_1).Hist_o \cup X$.
- T3 transitions in the specification state machines have (almost) this effect, so $A(s_1) \rightarrow_{spec} A(s_2)$. The difference is that replies do not get sent to the client, but this is equivalent, from client's perspective, to the replies being sent and then lost by the unreliable channel.

Failure of Internal Server S

- Upon detecting failure of internal (i.e., not head or tail) server S, the master removes S from the chain by telling S^- of its new successor and S^+ of its new predecessor.
- Problem: If that's all we do, UPI may be violated. In example below, H_C is not a prefix of H_A, violating UPI.



Remove B from chain

Failure of Other (Internal) Server S

- To fix this, S^- should forward “in-transit” updates to S^+.
- The head labels every update with a sequence number.
- The following message sequence accomplishes this.

1. Master sends S^+ a notification that S crashed.
2. S^+ sends master an acknowledgement containing the sequence number *sn* of the last update it received.
3. Master sends S^- a notification that S crashed containing *sn*.
4. S^- sends S^+ the sequence $\bar{r}$ of updates that S^- has received and that have sequence number greater than *sn*.

- To simplify computation of $\bar{r}$, S^- suspends processing of updates after receiving message 3 until sending message 4.

Failure of Other (Internal) Server S

- To accomplish 4., each server S other than the tail maintains sequences Sent^i_o of updates for o that S forwarded to some successor (hence its new successor should also receive, if it hasn’t already) and are possibly not yet processed by the tail (hence its new successor possibly didn’t receive).
  - When S forwards an update r for o: Sent^S_o.add(r)
  - When tail receives update r: send ack(r) to T^- (if any)
  - When S receives ack(r) for o: Sent^S_o.remove(r);
    send ack(r) to S^- (if any)
  - Problem: Propagation of ack(r) up the chain could be halted by a crash, and r would never get removed from some Sent sets. How to overcome this?

Acks Lost Due To Crashes

- Approach 1: Extend the algorithm to retransmit acks as needed after the chain is reconfigured due to a failure.
- Approach 2 (simpler):
  - The tail processes updates in sequence number order, so an ack for an update with sequence number *sn* can also serve as an can for all updates with lower sequence numbers.
  - Therefore, when S receives ack(r) for o, S can remove from Sent^S_o all requests with sequence numbers less than or equals to r's sequence number.
  - Acks lost due to crashes will not be retransmitted, but the requests they acknowledge will be removed from Sent^S_o anyway.

Inprocess Requests Invariant

- Using Sent^i_o, we can strengthen UPI to …
- Inprocess Requests Invariant: $i \le j \Rightarrow \text{Hist}_o^{\,i} = \text{Hist}_o^{\,j} \oplus \text{Sent}^i_o$
    - $r_1 \oplus r_2$ = merge of sequences of updates r_1 and r_2 in order consistent with the sequence numbers.

Failure of Other (Internal) Server S

- Let s_1 and s_2 denote the implementation state before and after these changes, respectively. Show: $A(s_1) \rightarrow_{spec} A(s_2)$.
- Any request received by S was also received by S^-, so removal of S does not cause any requests to be removed from $A(s_2).Pending_o$.
- If S^+ is not the tail, then $A(s_2).Pending_o = A(s1).Pending_o$ and $A(s_2).Hist_o = A(s1).Hist_o$, because no requests get processed by the tail during the reconfiguration.
- If S^+ is the tail, then $A(s_2).Pending_o = A(s1).Pending_o - \{X\}$ and $A(s_2).Hist_o = A(s1).Hist_o \cup \{X\}$, where X is the set of requests forwarded by S^- and processed by S^+ during the reconfiguration. T3 transitions in the specification state machines have this effect, so $A(s_1) \rightarrow_{spec} A(s_2)$.

Extending a Chain

- Failed servers should be replaced with new servers to maintain high availability.
- Simplest approach is to add new server at tail end.
- Let T^+ denote the new server.
- Initialization: $Sent^{T+}{}_o$ = empty sequence

  $Hist_o^{T+}$ = copy of $Hist_o^T$

  - This satisfies the Inprocess Requests Invariant.
- o may be large, so copying it to T^+ might take a while, so we allow T to continue processing requests during the copy.

Extending a Chain

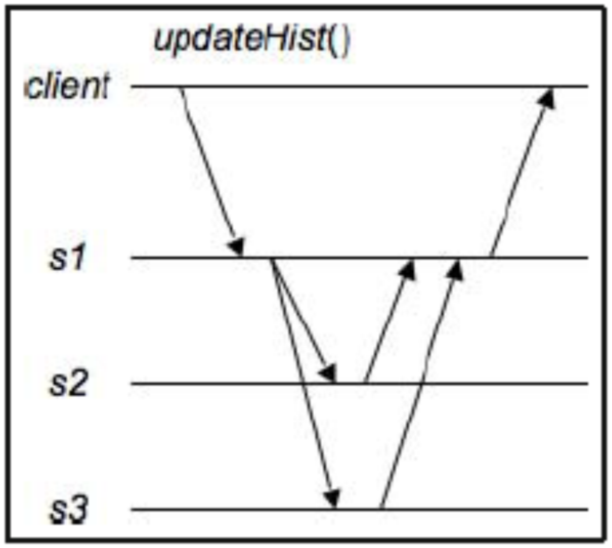
1. $T^+ \rightarrow$ Master: I'd like to join
2. Master $\rightarrow$ T: T^+ wants to join
3. T: start to store each new update it processes in Sent^T (like other non-tail servers are already doing)
4. $T \rightarrow T^+$: Hist_o^T for each object o.
5. $T \rightarrow T^+$: Sent^T (T^+ applies these updates but does not send replies to client, because T already did)
6. T: Upon sending "end of Sent^T", stop acting as the tail, i.e., stop sending replies to client
7. T^+: Upon receiving "end of Sent^T", start acting as the tail Note: no process other than T will send requests to T^+ yet.
8. $T^+ \rightarrow$ Master: I am the new tail
9. Master $\rightarrow$ Clients: T^+ is the new tail.

Extending a Chain

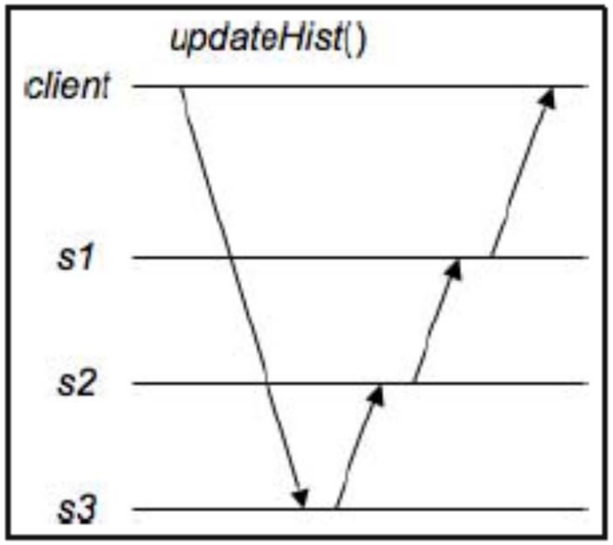
- If T fails during this process, T^+ waits until T^- has taken over as the tail, and then re-starts the chain extension protocol with T^-.
- Clarification: In implementation, T sends state of o piece by piece (it doesn't really send $Hist_o^T$) before sending $Sent^T$. If T processes an update u to part of o before sending that part of o to T^+, then T should not add u to $Sent^T$.

Primary-Backup

- Primary-backup uses a star-like, instead of chain-like, communication pattern between servers.
- In primary-backup, clients communicate with only one server (the primary), instead two (head and tail).



Primary-Backup



Chain Replication

Primary-Backup Protocol

- Query

1. C $\rightarrow$ P: query
2. P $\rightarrow$ C: reply

- Update

1. C $\rightarrow$ P: update.
2. P: compute state update and reply and send them to all backups.
3. P: wait for acknowledgment from all backups.
4. P $\rightarrow$ C: reply

C = client, P = primary

Primary-Backup: Failure Handling

- If primary fails, exactly one backup takes over as primary.
- A fixed priority ordering on servers can be used.
- All backups need to agree on which servers failed, so they can agree on new primary.
- As in chain replication, a consensus protocol, such as Paxos, is used to fault-tolerantly achieve this agreement.
- Primary might have failed after sending a state change to some but not all backups. New primary should query all servers to obtain most recent state change and forward it to all servers as needed, before processing new requests.

Performance Comparison of Primary-Backup and Chain Replication

- Chain replication has lower latency for queries.
- Suppose an update from one client arrives and a query from another client arrives immediately afterwards.
- In chain replication, the tail responds immediately to the query, while the head is processing the update.
- In primary-backup, the primary must process the update, send state change to backups, and wait for their acknowledgements, before processing the query.
- Chain replication has higher latency for updates.
- In chain replication, update propagates sequentially along chain, so latency is proportional to t, the number of servers.
- In primary-backup, the update is handled in parallel by all backups, so latency is independent of t.

Performance Comparison of Primary-Backup and Chain Replication

- Chain replication has higher throughput for workloads containing a mixture of queries and updates, because it splits the work of request processing among two servers, while primary-backup assigns all request processing to one server.