Probabilistic Consistency and Durability in RAINS: Redundant Array of Independent, Non-Durable Stores

Andy Huang and Armando Fox

Stanford University

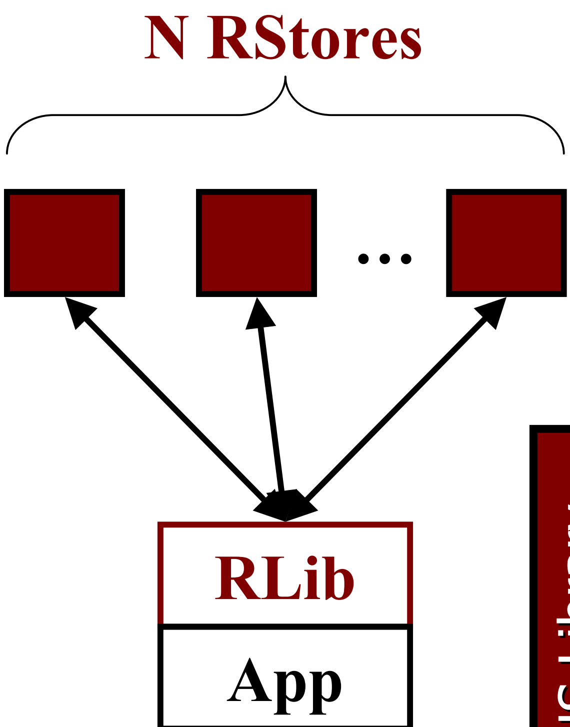
Motivation: [app needs] $\neq$ [storage guarantees]

- Some types of state in interactive Web applications don't require absolute consistency and durability
  - Examples: shopping cart state, session state
  - Characteristics: lifetime ~wks/mins, lost updates not critical
- This state is often stored in storage solutions that provide absolute guarantees (e.g., DBMS, DDS)
- Applications pay the cost of these policies even if they don't need the guarantees
  - Node failure recovery: some r/w operations are suspended
  - Scaling: downtime of some partitions and administration

Proposal: Relax consistency and durability

- Approach: Provide a storage solution that apps can tune for desired levels of consistency and durability
- Proposed Solution: Store state in a redundant array of independent, non-durable stores (RAINS)
  - Independent: No coordination (2P commit) among nodes
  - Non-durable: Node doesn't necessarily recover state on crash recovery (Web cache)
- Hypothesis: Storing data in a RAINS significantly reduces the costs of node failures and scaling
  - Fast recovery (no need to recover data)
  - Reads and writes allowed during recovery and scaling

RAINS Hashtable Overview



RAINS Store

- Unreliable, non-durable hashtable (RLib can't tell if `put` succeeds or `get` returns up-to-date value)
- Consistency and durability mechanism: write/read N RStores

RAINS Library

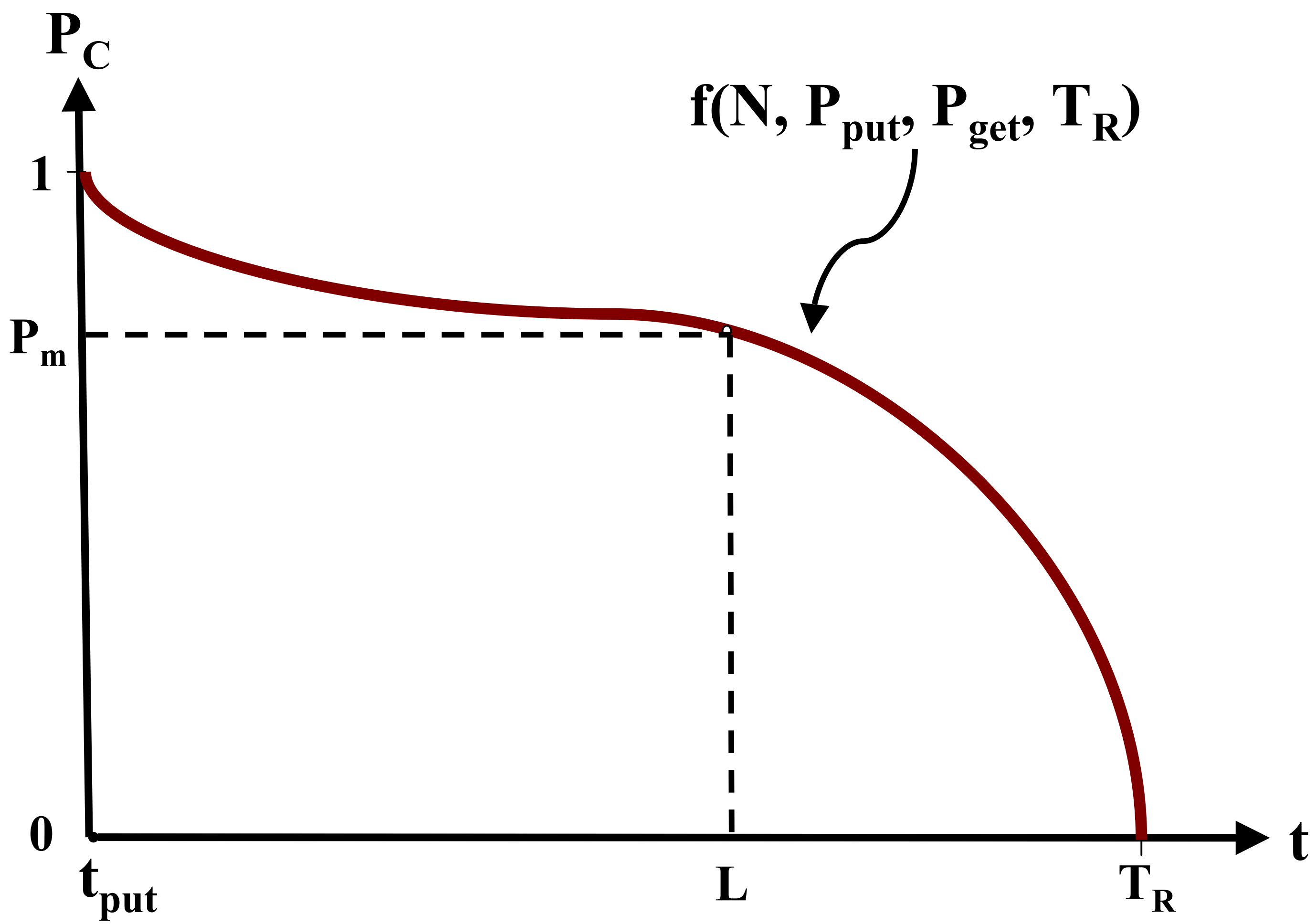
- Presents hashtable-like API to App
- Constructor: `hashtable(L,`P_m`)` L=lifetime, P_m=min. P(consistency)
- RLib uses L and P_m to determine N
- Uses some policy to determine return value of `get` (e.g., majority)

Probabilistic Model for Determining N

- Given: `hashtable(L,`P_m`)`
  - L = lifetime
  - P_m = min. consistency probability = min(P_C)
- Assume we can calculate these values:
  - P_{put} = P(put succeeds)
  - P_{get} = P(get returns a valid value if it exists)
  - T_R = frequency of RStore reboots (chosen s.t., crashes are very unlikely between reboots - based on TTF distribution)
- Find:
  - N = # RStores to write and read

    $= f(L, P_m, P_{put}, P_{get}, T_R)$; s.t., $\forall\ t \leq L, P_c \geq P_m$

P(consistency) Decreases Over Time



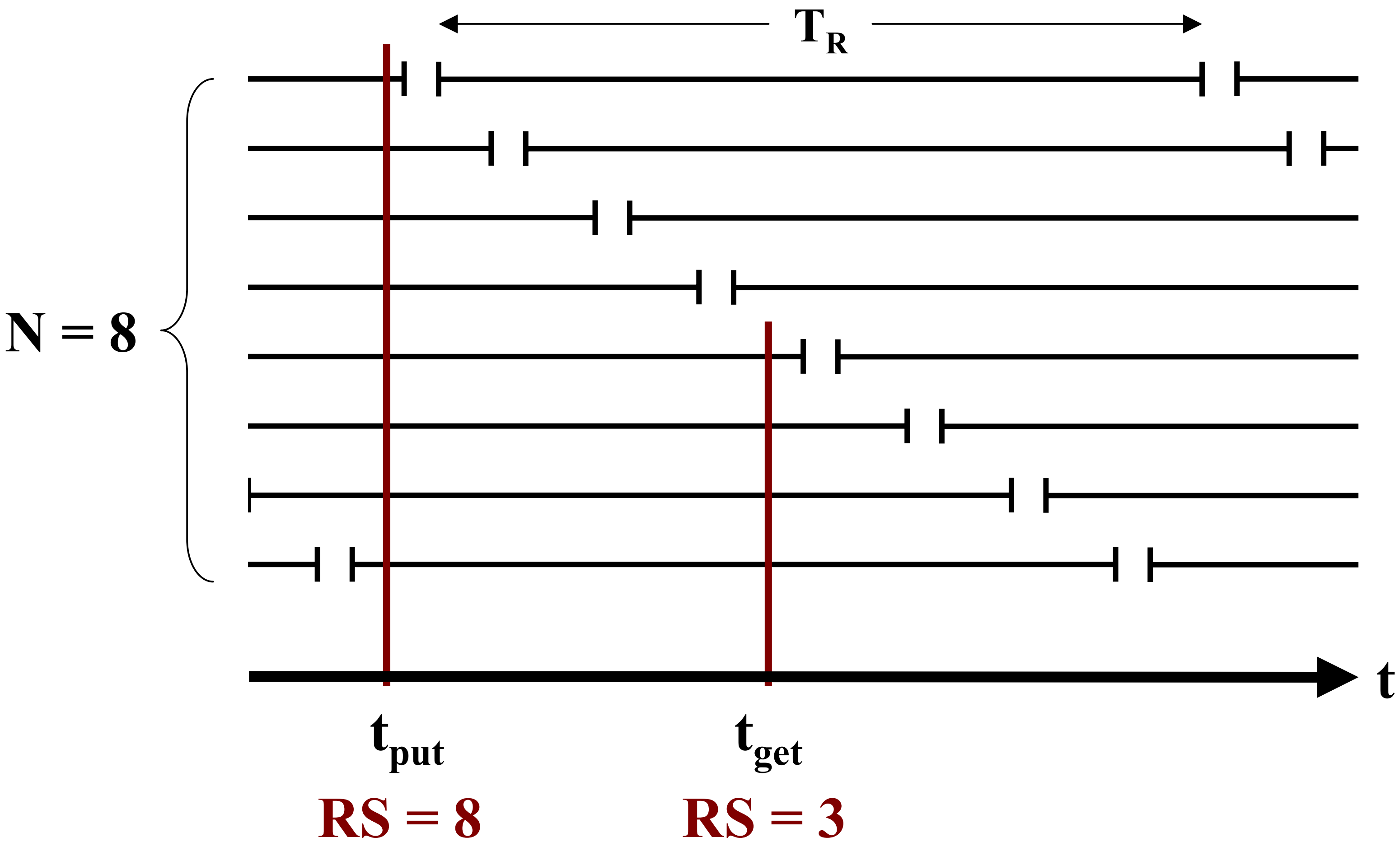
Assumptions

1. Network is perfect and node failures are independent → individual `put` and `get` operations are independent of each other
2. Data isn't recovered on node recovery
3. Rolling reboots: reboots spaced in regular intervals
4. Write a distinct value on each `put`
5. RLib can detect corrupt values (e.g., using CRC)

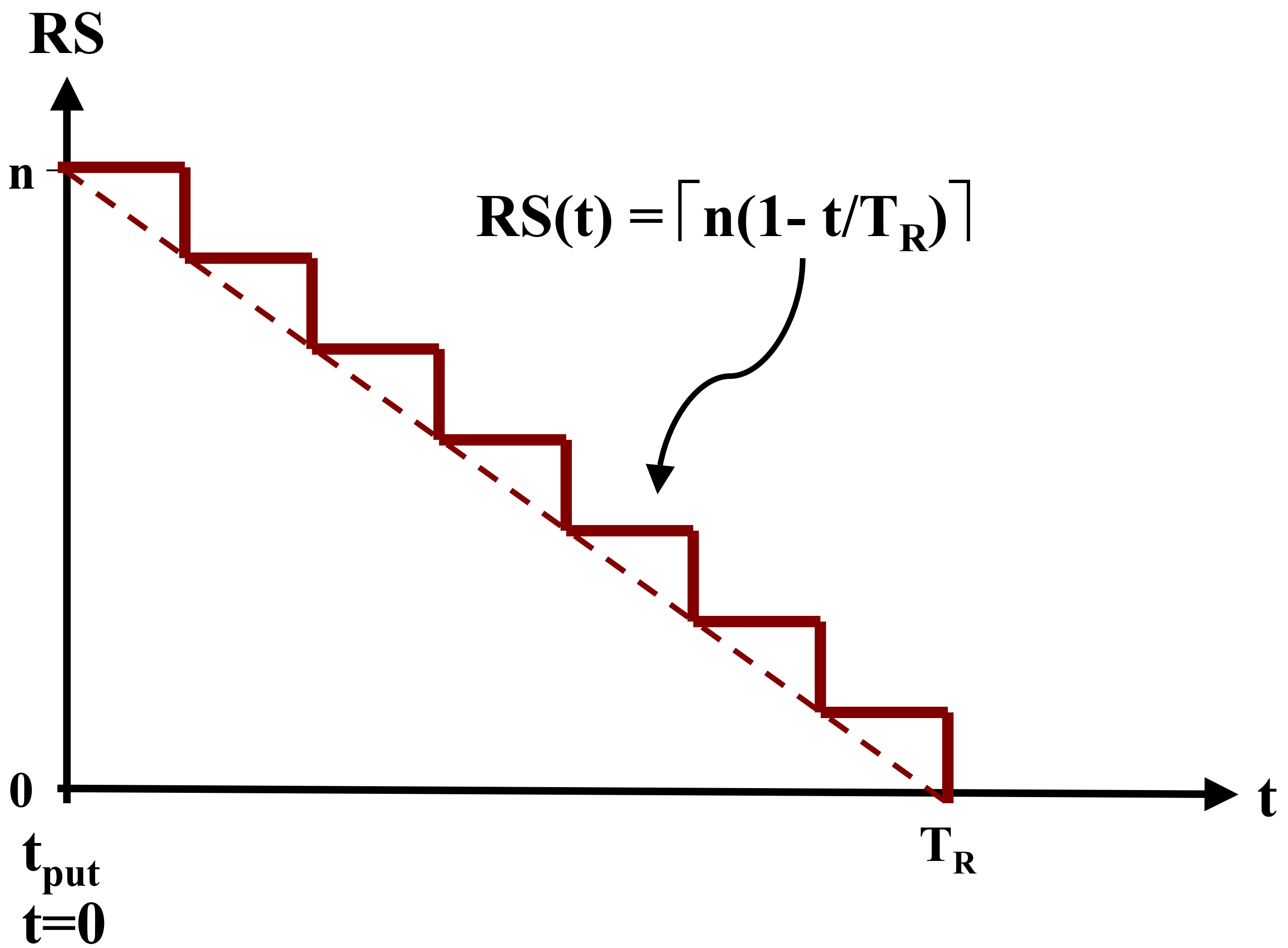
Read Set – RS(t)

- $RS(t)$ = [# RStores that have data from $t=t_{put}$]
- A2 (data isn't recovered on recovery)
  - → $RS(t)$ is monotonically decreasing
  - $RS(t)$ = [# RStores that haven't been rebooted since $t=t_{put}$]
- A3 (reboots spaced in regular intervals)
  - → $RS(t)$ is a step function bounded by a linear function (decreasing at a constant rate)

Read Set Example: N=8



Read Set Function



Possible Read Outcomes

1. Valid, up-to-date value (U): $P_U = P_{put} \times P_{get}$
2. Valid, stale value (S): $P_S = (1 - P_{put}) \times P_{get}$
   - A4 (each `put` value is distinct)
   - → possible that S = U, so P_C errs on the pessimistic side
3. Fail - timeout or corrupt value (F): $P_F = 1 – P_{get}$
   - A5 (RLib can detect corrupt values using CRC)
   - → corrupt values and timeouts can be conflated
4. Key not found (N)
   - If ∃ a valid value (U/S), assume N is from rebooted RStore
   - → N is not part of the Read Set.

Policies for Determining Return Value of `get`

- Let: n = RS(t)
- Most up-to-date time stamp
  - $P_C = 1 - (1 - P_U)^n$
- Majority: $U_n > n/2$
  - Let $X_i = 1$ if i^{th} value is up-to-date
    $= 0$ if i^{th} value is fail or stale
  - Let $U_n = X_1 + X_2 + \dots + X_n$ (binomial distribution)
  - $P(U_n = i) = \binom{n}{i} P_U^{\,i} \times (1 - P_U)^{n-i}$
  - $P_C = P(U_n > n/2) = \sum^{n}_{i=n/2+1} P(U_n = i)$
- Majority over Read Set: $U_n > S_n$

Questions

- How big is the win?
  - Can we achieve throughput as good as or better than DDS?
  - How cheap can we make failures?
- What is the cost?
  - Can we achieve decent levels of consistency and durability (even with small clusters)?
  - Are there an interesting set of applications that can tolerate relaxed consistency and durability?

Conclusion

- Problem: Storage solutions force applications to pay the cost of absolute consistency and durability even if they aren't needed
- Proposal: Store state in RAINS, which allows applications to specify the desired level of consistency and durability
- Hypothesis: Relaxing the levels of consistency and durability will drastically decrease the costs of node failures and scaling