Distributed state trade-offs

Ousterhout. The Role of Distributed State.

Last class:
What is distributed state?

- Snapshot of the system in time: what everyone is doing at that moment
- Birds of flock metaphor
- **Distributed <u>global</u> state: State at process + State of channels**
- Snapshot starts … snapshot ends
    - Resulting snapshotted state is in-between (potential state that is reachable from start, and can reach end state)

Distributed state in this paper (why so different?)

- Dist. State in previous paper is the same!? We don't capture it, we only use aspects of it.
- Prev. Paper — distributed state: state at node + state in channels. Both states are user defined.
- Why did we snapshot state? One is for debugging. Second is for reliability — re-create the system from the snapshot.
- In this paper — *"information retained in one place that describes something, or is determined by something, somewhere else in the system".* (Template definition)
    - Don't care about internal states of processes, nor channels (consider state after msgs are reflected / processed by the node)
    - Simpler! Easier to reason about: better for reasoning about the design of the system!
    - Not as generally useful — you can't re-create the system from this state
    - Did he really define .. anything!? What does this def. really mean?
- Distribute state is really up to you define! It's a subjective thing. "Statelessness and statefulness" — these are important, but highly subjective (without state there is no computation).

How does NFS work?

- (see whiteboard)
- Designed for simplicity, and ability to restart the server
- Server stateless but clients do retain state (mapping of files to ID handles, and file cache)
- Retry mechanism works for both msg loss and server crashes
- Idempotent msgs yield same result: retries aren't handled specially. There is no diff. Between retry and original msg (on the client side!)
- But, no consistency semantics per se (not formally defined, wait and see, race conditions on multiple writers)
- High msg cost, disk write through inefficient on server (client blocks waiting on server disk).
    - Statelessness reduces a client operation to performance of the disk on the server. Coupling perf of client ops to disk perf (this is bad)
    - Disks are getting faster much slower than processors (and networks) => the above is a terrible strategy long term. You *need* state for performance.

How does NFS work?

- NVRAM to the rescue?
- NVRAM ~ RAM that survives failures (e.g., capacitor that flushes state to disk on failure)
- NVRAM replaces disk b/c it is faster than disk and it survives failures, so you can use it during normal operation instead of disk

How does Sprite work?

- (see whiteboard)
- Server stateful: knows which file is opened by which client
    - Server reconstructs state when clients reconnect with *reopen*
- Client stateful: has a cache of dirty blocks (written data)
    - Clients flush state on close, or periodically
- Messages are not idempotent
- Recovery complex
- Sequential consistency with multiple writers
- Performance is much much faster (due to multiple levels of caching on both read and write paths)

Tradeoffs of dist. state

- **The bad**
- Reliability requires distributed state or complete statelessness, but trades off with opportunities for efficiencies.
- Dist state has a storage cost (duplication: diff nodes store same or similar information e.g., caching)
- Dist state is more complex. If you want to rely on it, you have to trust it, to trust it you need to know what it represents and its consistency
    - These require extra energy/complexity to maintain (more protocols)
    - Dist state produces more corner cases (failures)

Tradeoffs of dist. state

- **The good**
- More fault tolerant (redundant information)
- More independent views on state means more chance for a reliable view when adversaries are in the system ~ byzantine fault tolerance
- Higher performance! I can cache state. I can move/maintain state closer to clients. More opportunities for parallelism (with more nodes).
- **You can avoid disks** b/c distributed state is tolerant of node failures.

Next class

- Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. CSUR 1990.

- State machine replication (fault tolerance) abstraction

- Key paper for reasoning about SMR