

# Persistent Actor State

Principles of Reactive Programming

Roland Kuhn

# Persistent Actor State

Actors representing a stateful resource

- ▶ shall not lose important state due to (system) failure
- ▶ must persist state as needed
- ▶ must recover state at (re)start

# Persistent Actor State

Actors representing a stateful resource

- ▶ shall not lose important state due to (system) failure
- ▶ must persist state as needed
- ▶ must recover state at (re)start

Two possibilities for persisting state:

- ▶ in-place updates
- ▶ persist changes in append-only fashion

# Changes vs. Current State

Benefits of persisting current state:

- ▶ Recovery of latest state in constant time.
- ▶ Data volume depends on number of records, not their change rate.

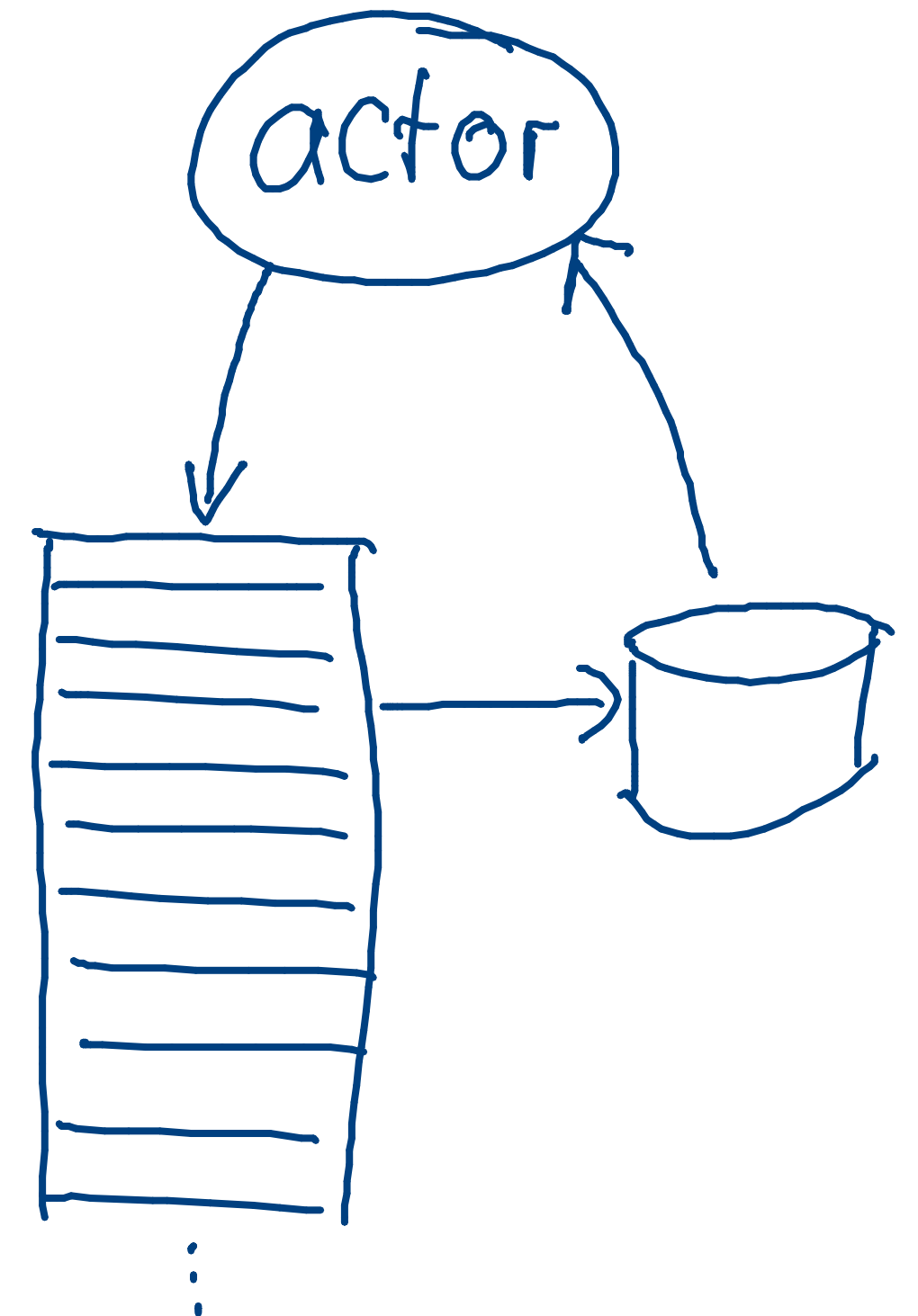
# Changes vs. Current State

Benefits of persisting current state:

- ▶ Recovery of latest state in constant time.
- ▶ Data volume depends on number of records, not their change rate.

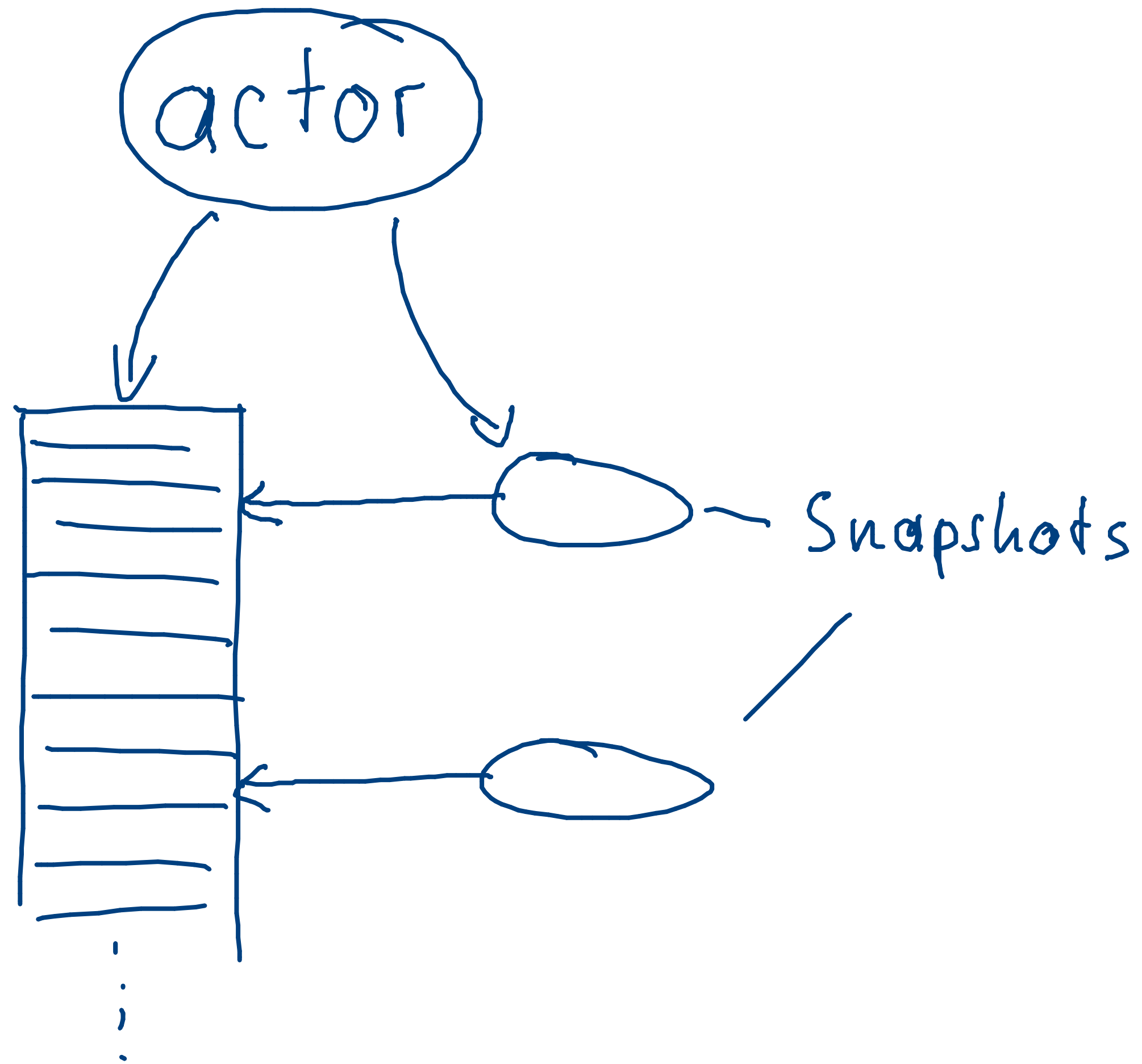
Benefits of persisting changes:

- ▶ History can be replayed, audited or restored.
- ▶ Some processing errors can be corrected retroactively.
- ▶ Additional insight can be gained on business processes.
- ▶ Writing an append-only stream optimizes IO bandwidth.
- ▶ Changes are immutable and can freely be replicated.



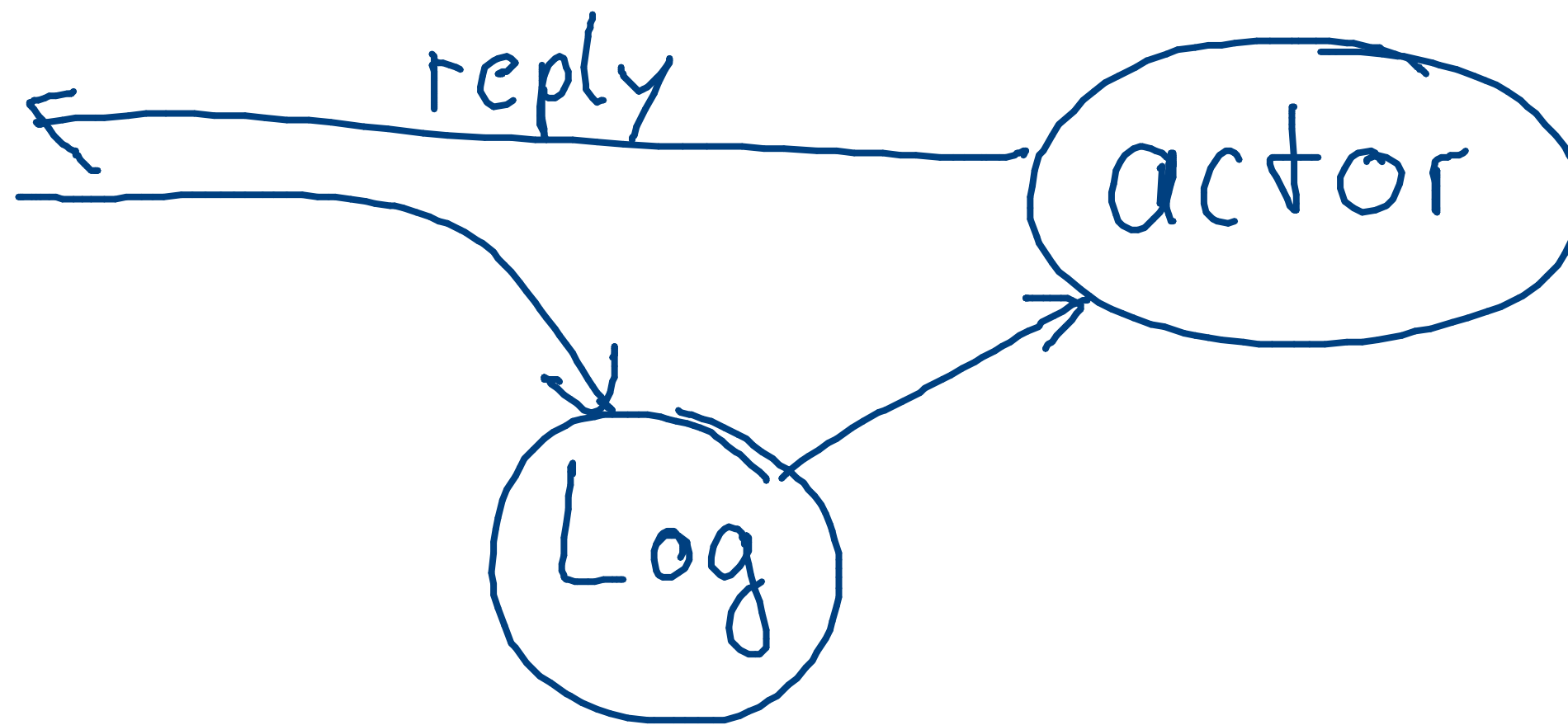
# Snapshots

Immutable snapshots can be used to bound recovery time.



# Command-Sourcing

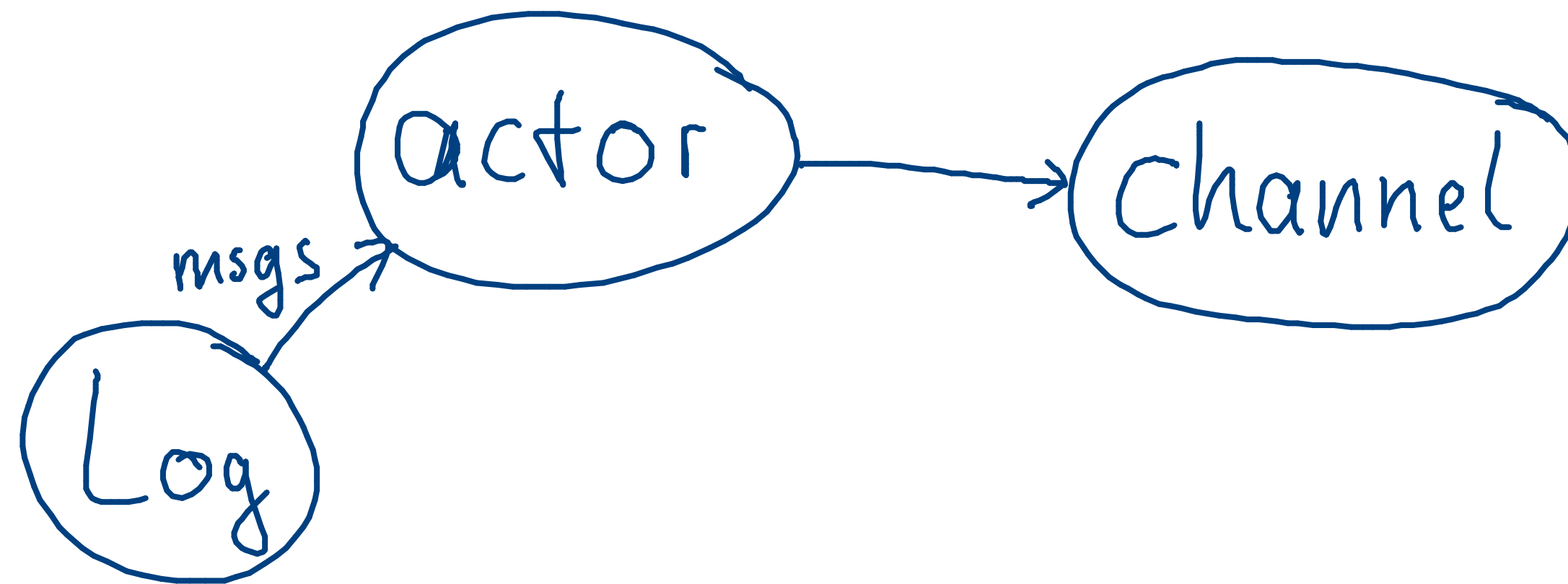
**Command-Sourcing:** Persist the command before processing it, persist acknowledgement when processed.



# Commands and Channels

During recovery

- ▶ all commands are replayed to recover state.
- ▶ a persistent Channel discards messages already sent to other actors.



# Event-Sourcing

**Event-Sourcing:** Generate change requests (“events”) instead of modifying local state; persist and apply them.

# When to Apply the Events?

- ▶ Applying after persisting leaves actor in stale state.

# When to Apply the Events?

- ▶ Applying after persisting leaves actor in stale state.
- ▶ Applying before persisting relies on regenerating during replay.

# When to Apply the Events?

- ▶ Applying after persisting leaves actor in stale state.
- ▶ Applying before persisting relies on regenerating during replay.

Trading performance for consistency:

- ▶ Do not process new messages while waiting for persistence.

# The Stash Trait

```
class UserProcessor extends Actor with Stash {  
  var state: State = ...  
  def receive = {  
    case NewPost(text) =>  
      emit(PostCreated(text), QuotaReached)  
      context.become(waiting(2), discardOld = false)  
  }  
  def waiting(n: Int): Receive = {  
    case e: Event =>  
      state = state.updated(e)  
      if (n == 1) { context.unbecome(); unstashAll() }  
      else context.become(waiting(n - 1))  
    case _ => stash()  
  }  
}
```

# Summary

- ▶ Actors can persist incoming messages or generated events.
- ▶ Events can be replicated and used to inform other components.
- ▶ Recovery replays past commands or events; snapshots reduce this cost.
- ▶ Actors can defer handling certain messages by using the Stash trait.