Compact Resettable Counter through Causal Stability

Georges Younes Paulo Sérgio Almeida Carlos Baquero

April 22, 2017

EUROSYS - PaPoC'17

Outline

Motivation

Motivation

- Satisfy user requirements -> developers must be able to compose complex data types together
  - We can achieve that using maps, where all other data types can be embedded within them, including maps themselves
- It is not that easy...
  - The problem with embedded CRDT counters
- Or is it?

Problem Definition

Problem definition

Maps need to support the update and removal of entries:

- entries: nested data types like counters, flags, registers… or even maps
- Creating and updating an entry call the entry's update function
- Removing an entry is done more carefully: We can't directly delete an entry. Why?

In replica **A** tries to delete the entry (delete a counter *ctr*) and concurrently replica **B** is updating the same entry (increment *ctr*)

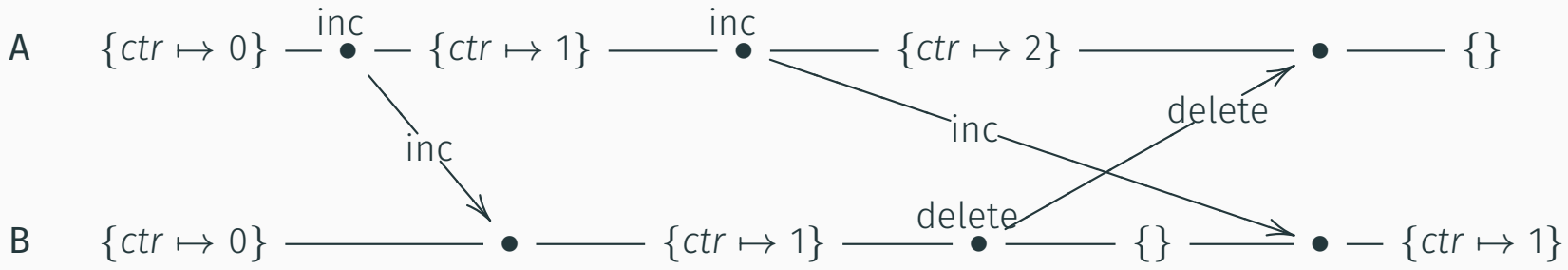


Figure 1: Deleting the entry (without careful checking) leads to inconsistency

Problem definition

- Maps need more careful checking on removal of entries:
  - Removing an entry: first reset the entry, then if entry's state is $\bot$ delete it from map
- Check Antidote's Resettable Map under `https://github.com/SyncFree/antidote_crdt/blob/master/src/antidote_crdt_map_rr.erl`

Problem definition

- We use a *Reset* semantics where a Reset **only** affects local **seen** operations:
  - Other *Reset* semantics could be: also affecting operations concurrent to the *Reset* operation
  - *Reset* should affect a subset of the operations: only ops that are in the causal past of its invocation
  - Any eventual operations that are concurrent to a *Reset* operation, should not be affected
- The state of current counter design is an integer
  - It is impossible to reset a subset of the operations as the state being single integer does not permit that

Problem definition

We illustrate the anomalies with 2 examples using 2 different *Reset* implementations that can operate on integers

In the first example, we implement the *reset* function as to set the counter to 0

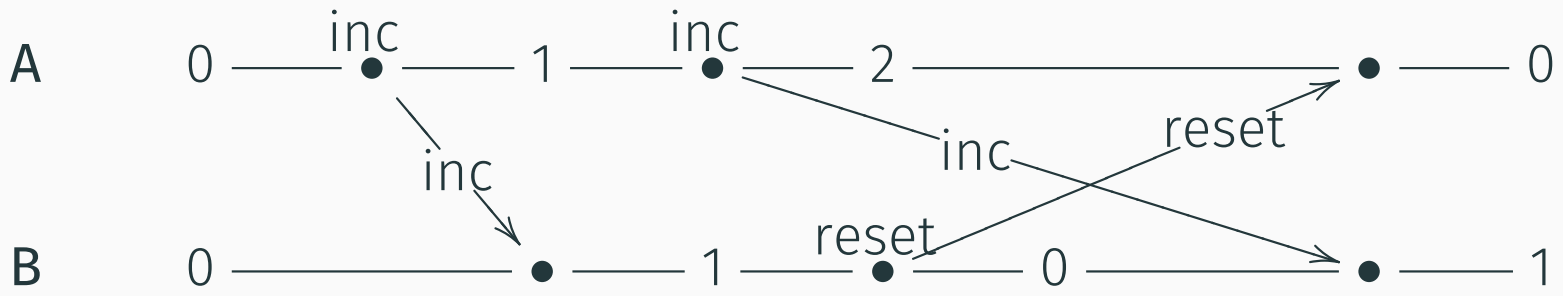


Figure 2: This implementation leads to inconsistency

Problem definition

In the second example, we implement the *reset* function as to decrement the counter by its current value

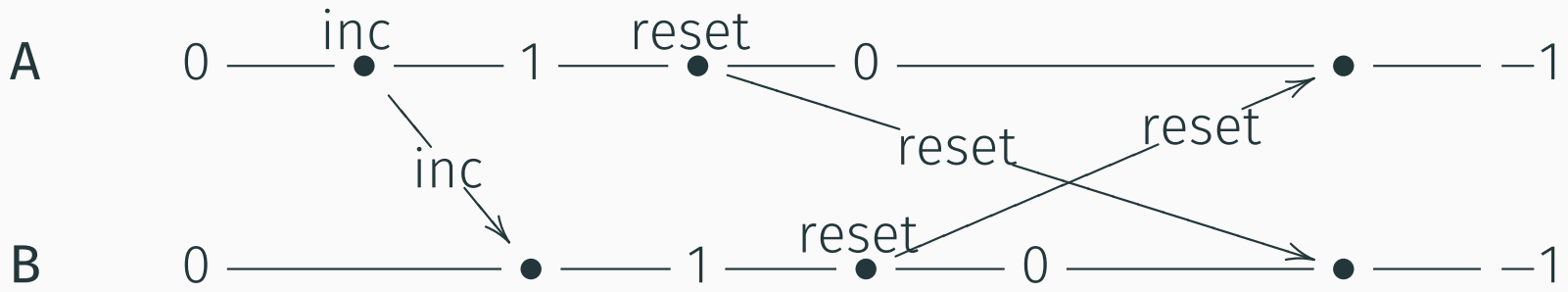


Figure 3: This implementation leads to an incorrect behavior

Naive Solution

Naive solution

As the problem is that the state is reduced to one integer, one solution is to:

- Tag each *inc* (also *dec*) operation with a timestamp
- The state would be the set of these operations
- A *query* would return the sum of *inc* operations - sum of *dec* operations
- A *reset* operation would reset all operations with timestamp in its causal past

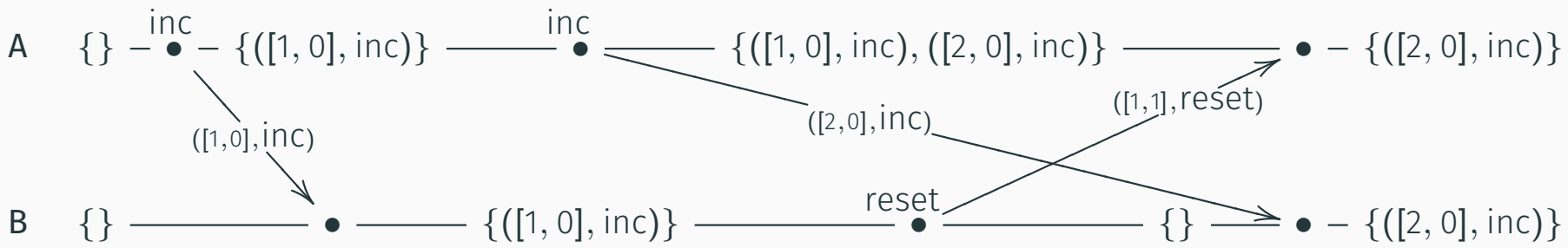


Figure 4: Example of a Naive Resettable Counter

Naive solution

It works, but it is still naive solution:

- The state keeps growing linearly with new increments and decrements
- Can we garbage collect? How?
- Sit back, relax and meet Causal Stability

Causal Stability

Causal Stability

Definition. *A timestamp t, and corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamp $t' > t$*

Stability can be locally detected by tracking in each node the last timestamps from each other node.

In the next slides, we present an example illustrating how causal stability works

Compact Solution

Compact Solution

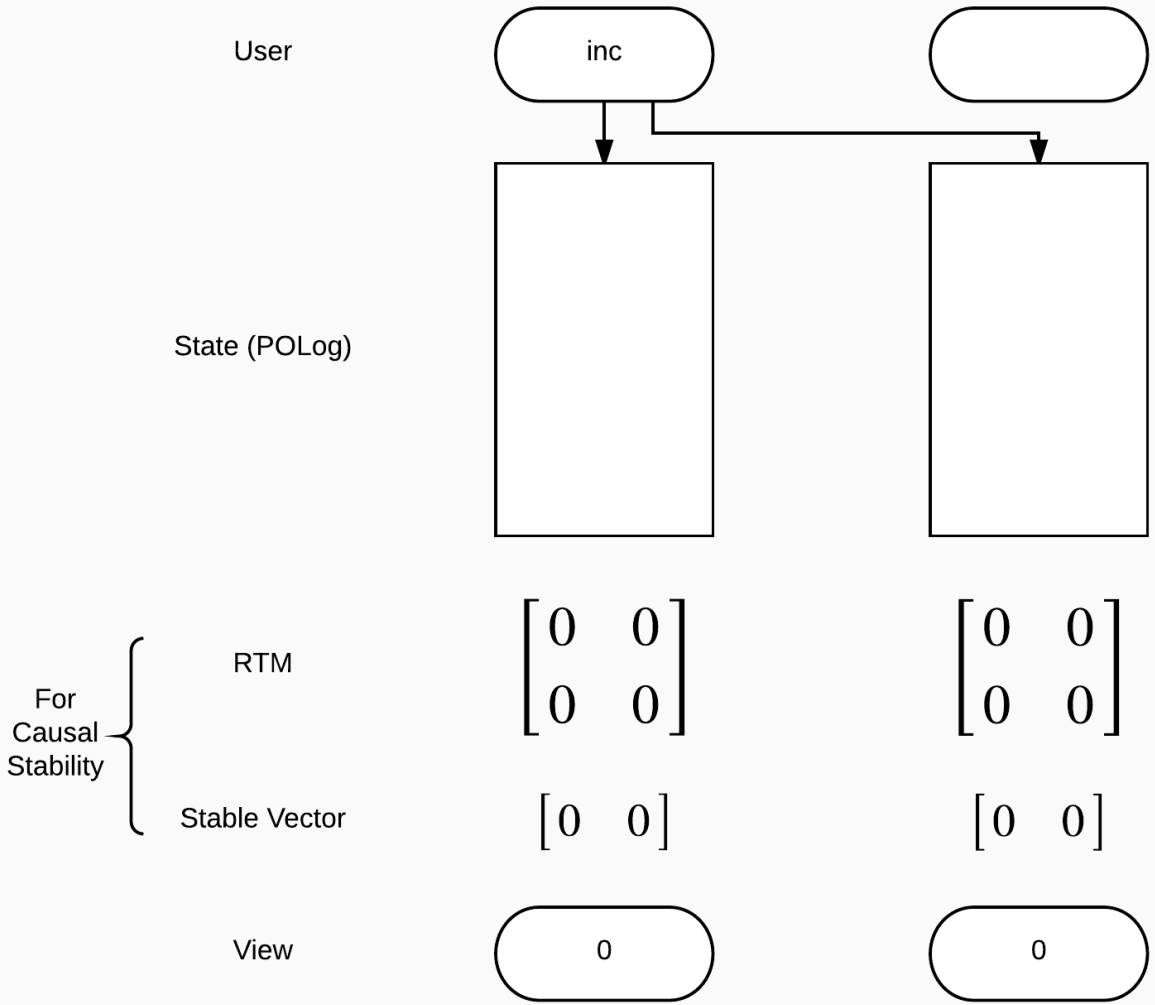
So our **compact solution** is basically applying the compaction technique allowed by **causal stability** over the **naive resettable counter**

Compact Solution

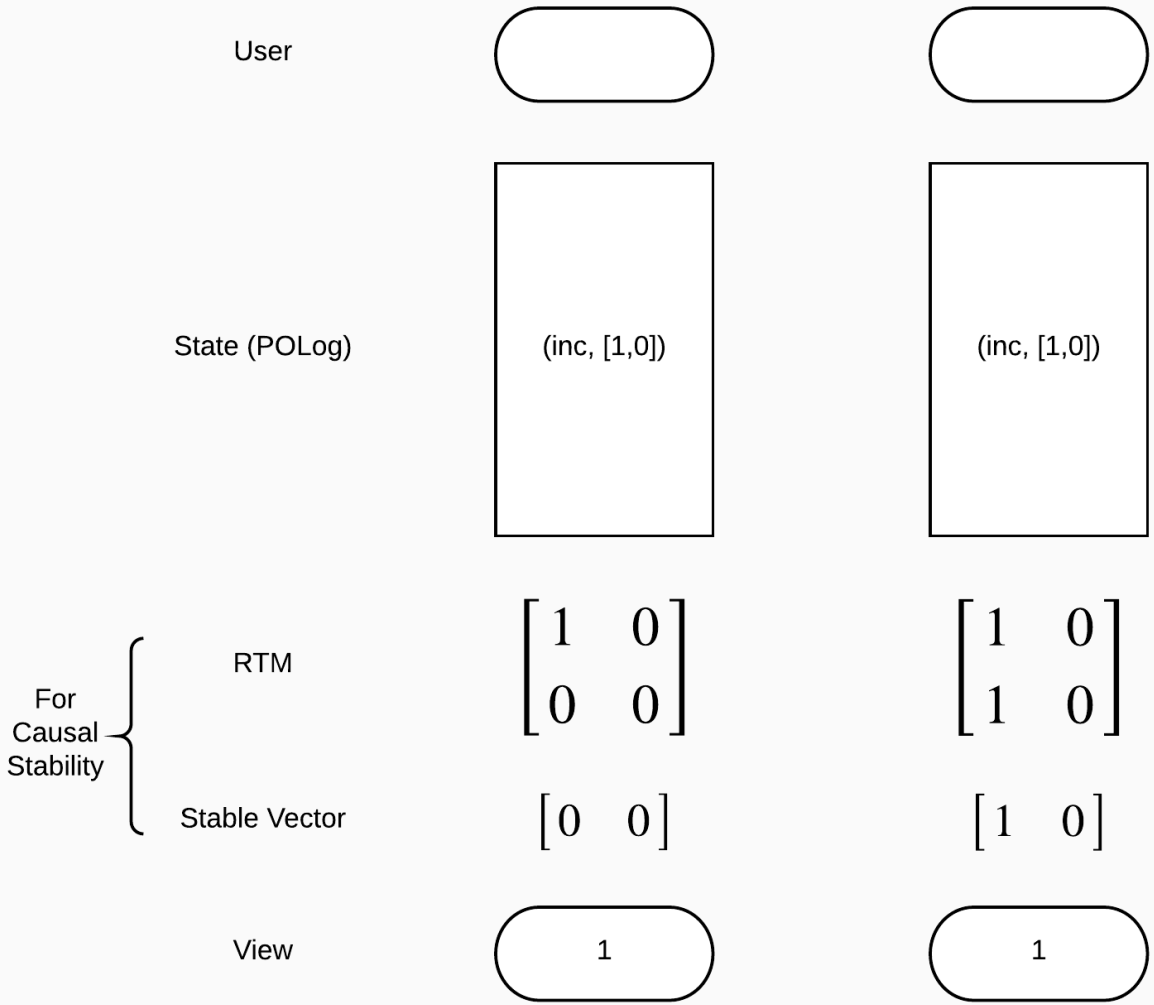
In this example:

- We use vector clocks as timestamps
- We show causal stability between 2 replicas
- We introduce a data structure called *RTM* (Recent Timestamps Matrix) that stores the knowledge of delivered messages: This *RTM* is updated locally at each node upon message sending/delivery
- We introduce a data structure called *Stable Vector* that is the point-wise minimum of all the vectors in the *RTM*
- A message is stable if its *VC* < *Stable Vector*

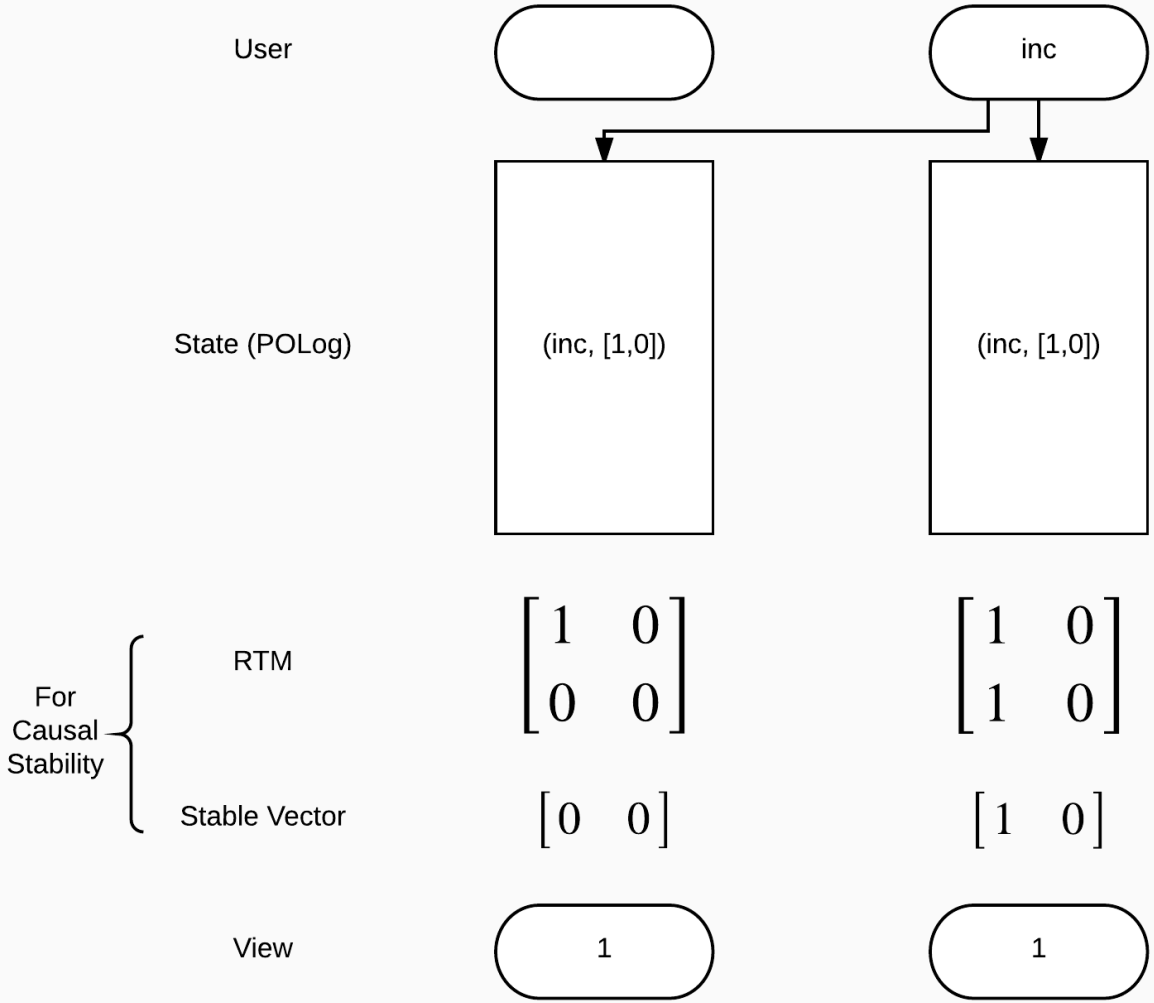
Compact Solution



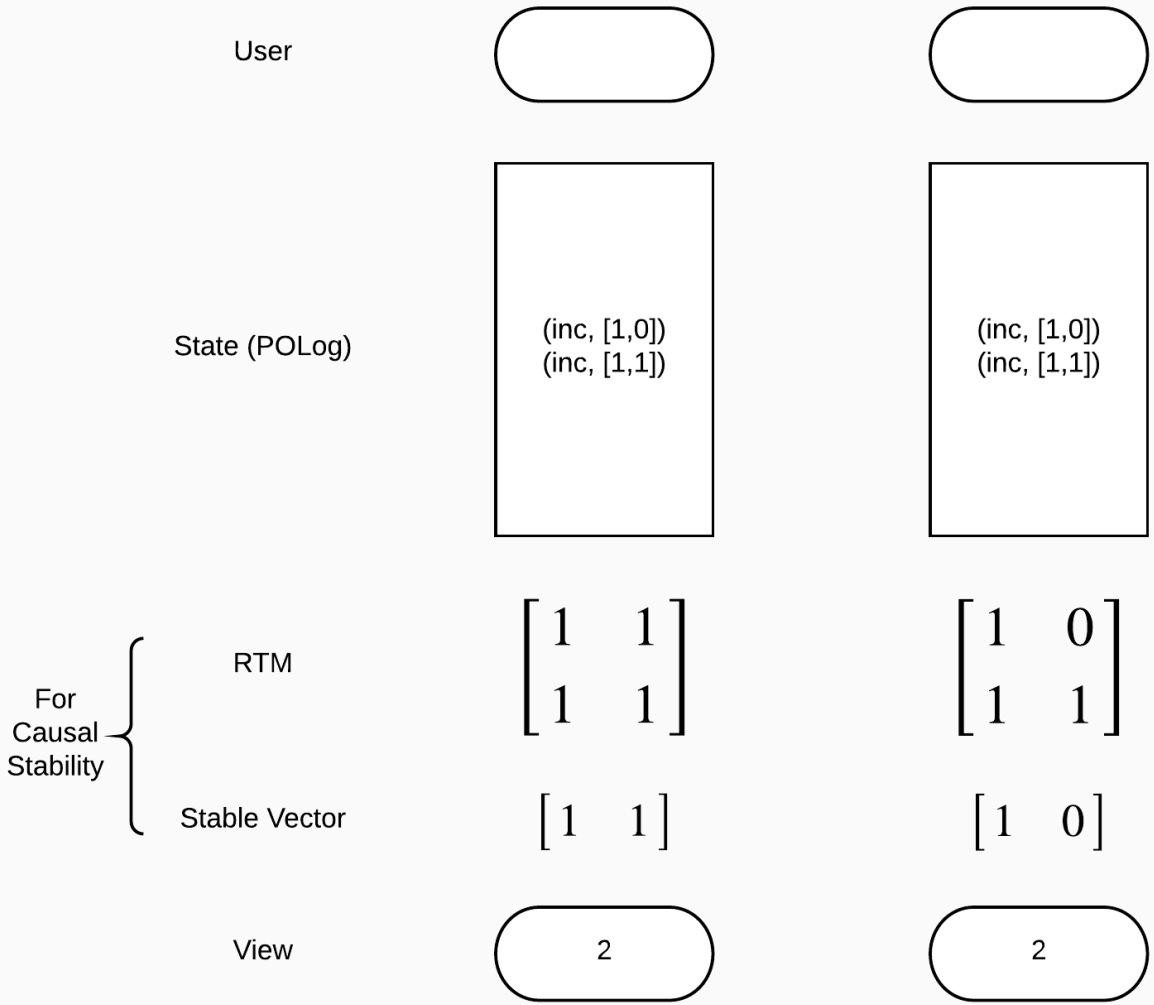
Compact Solution



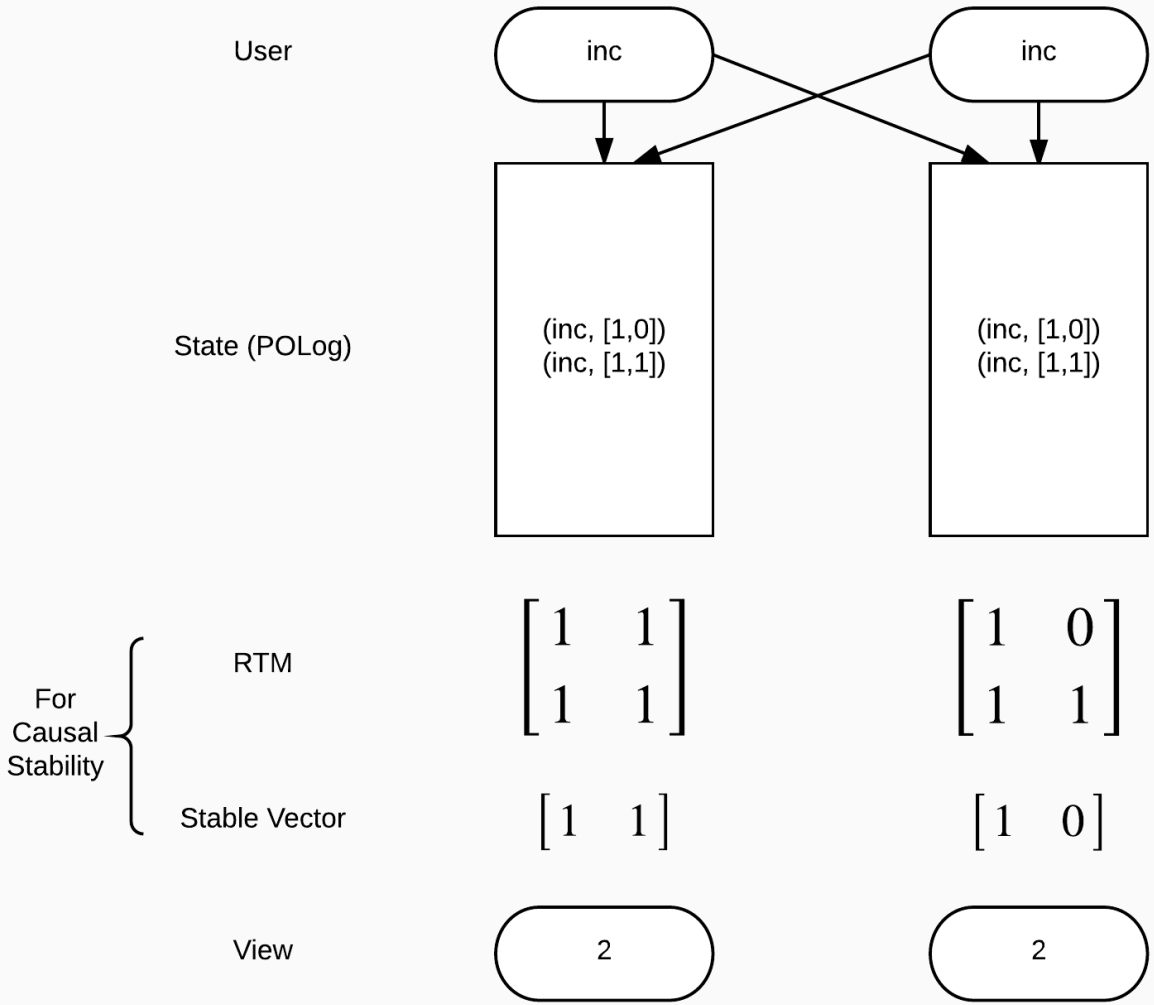
Compact Solution



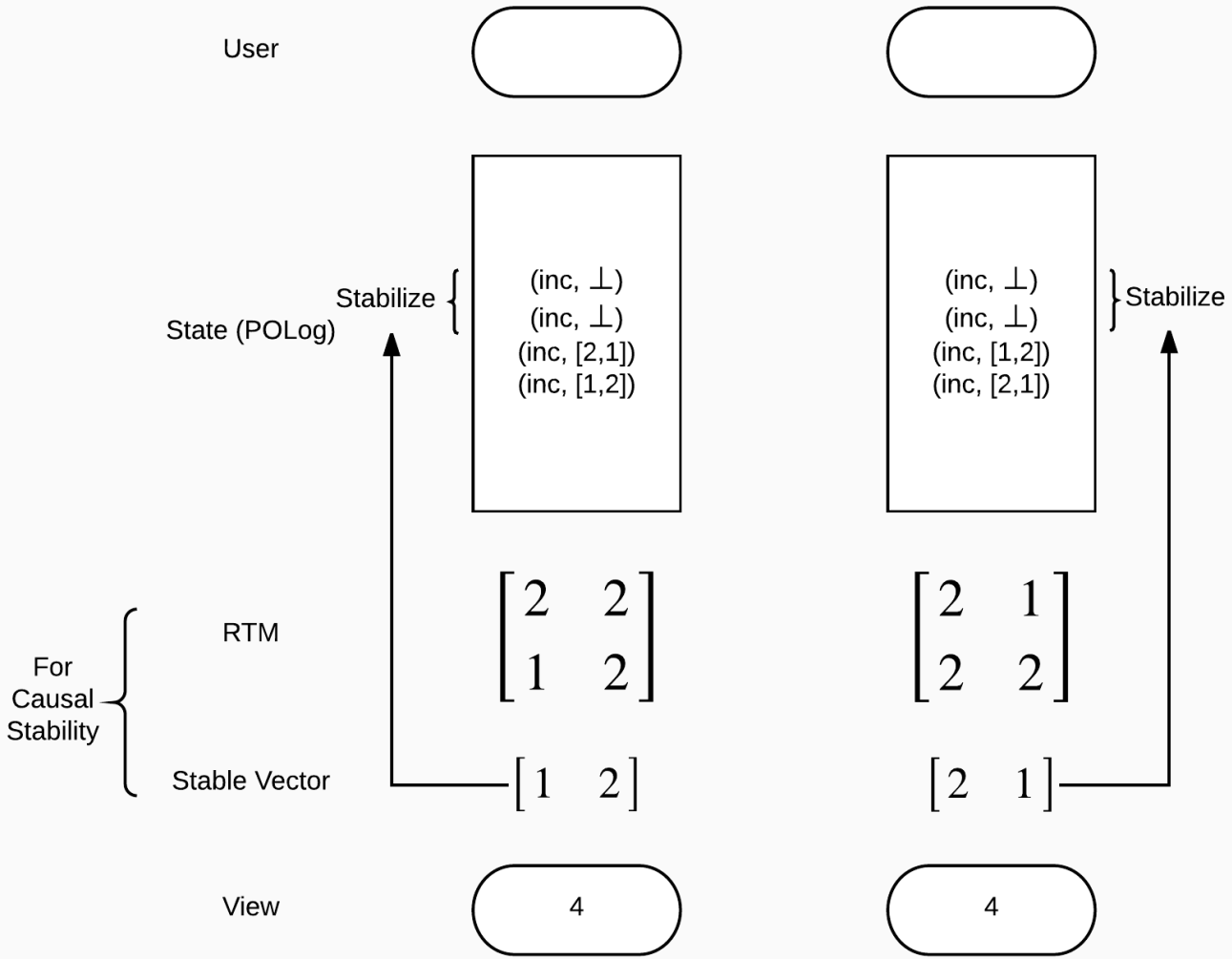
Compact Solution



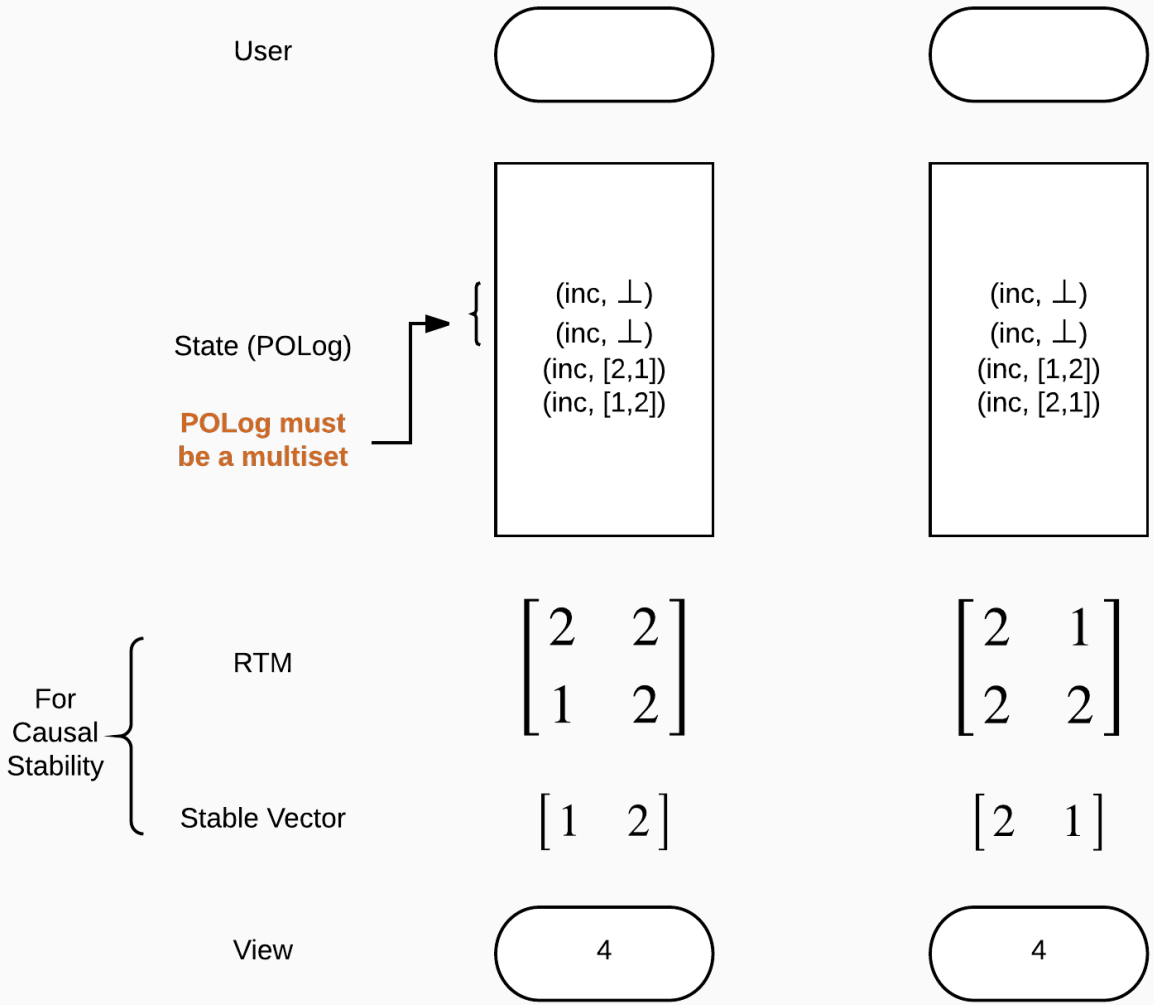
Compact Solution



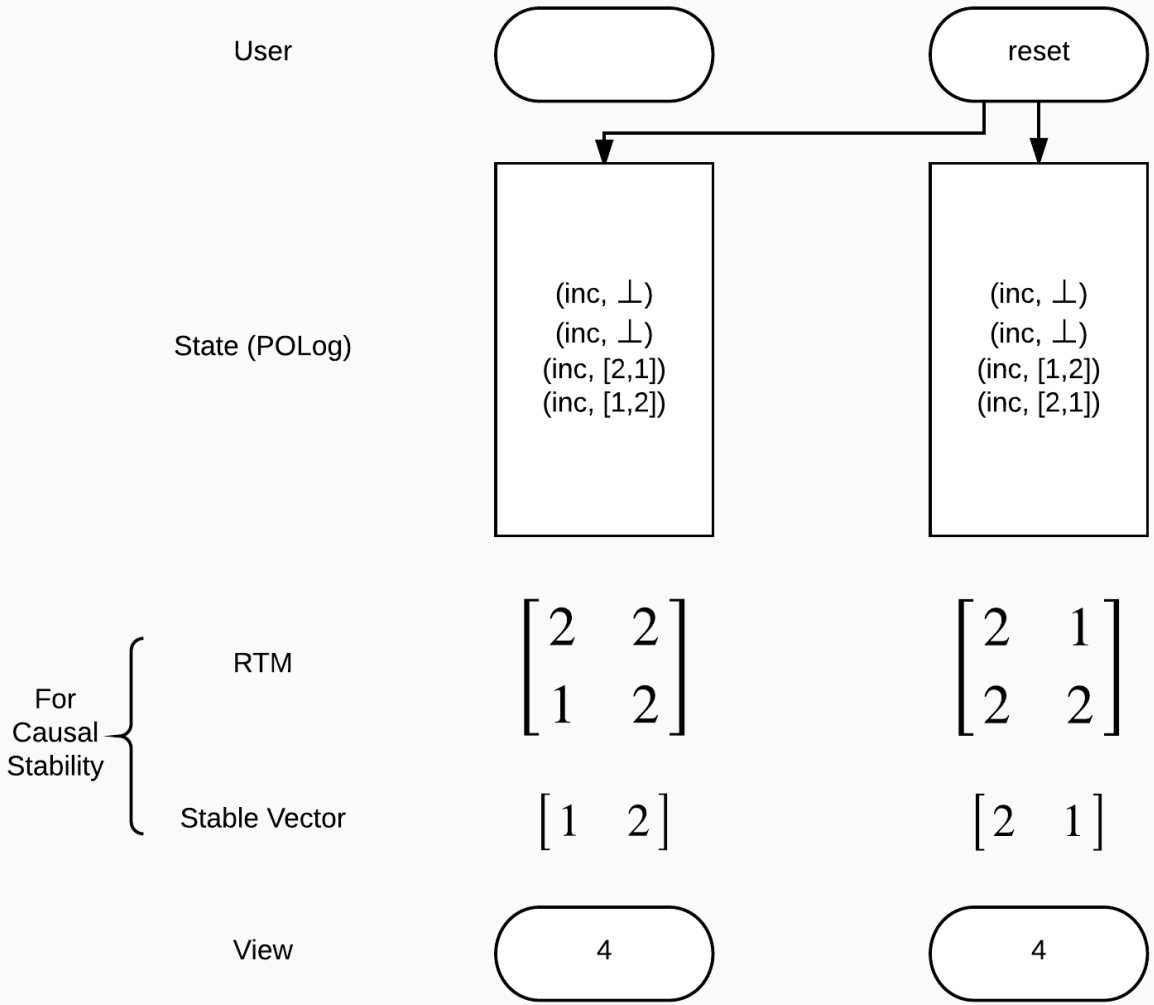
Compact Solution



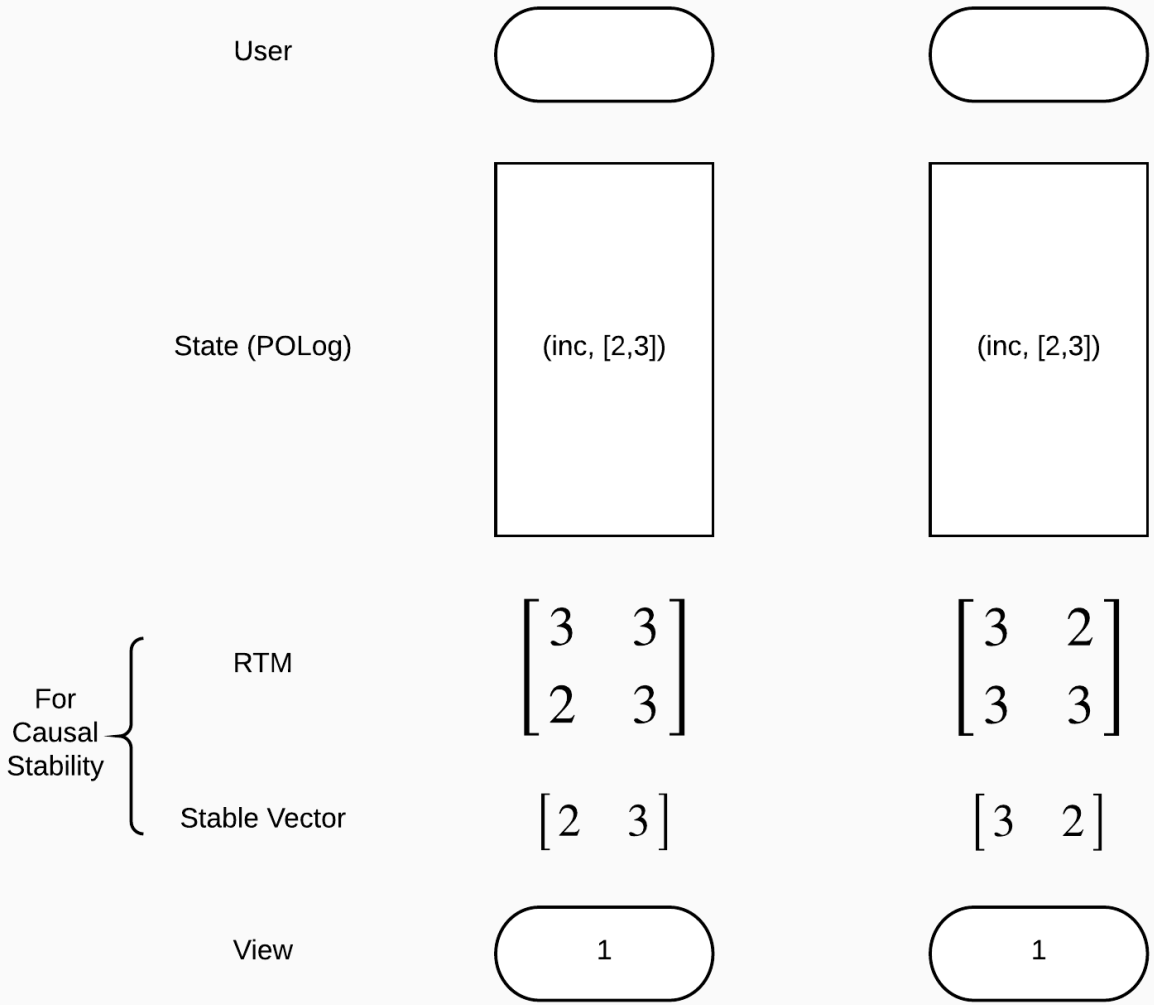
Compact Solution



Compact Solution

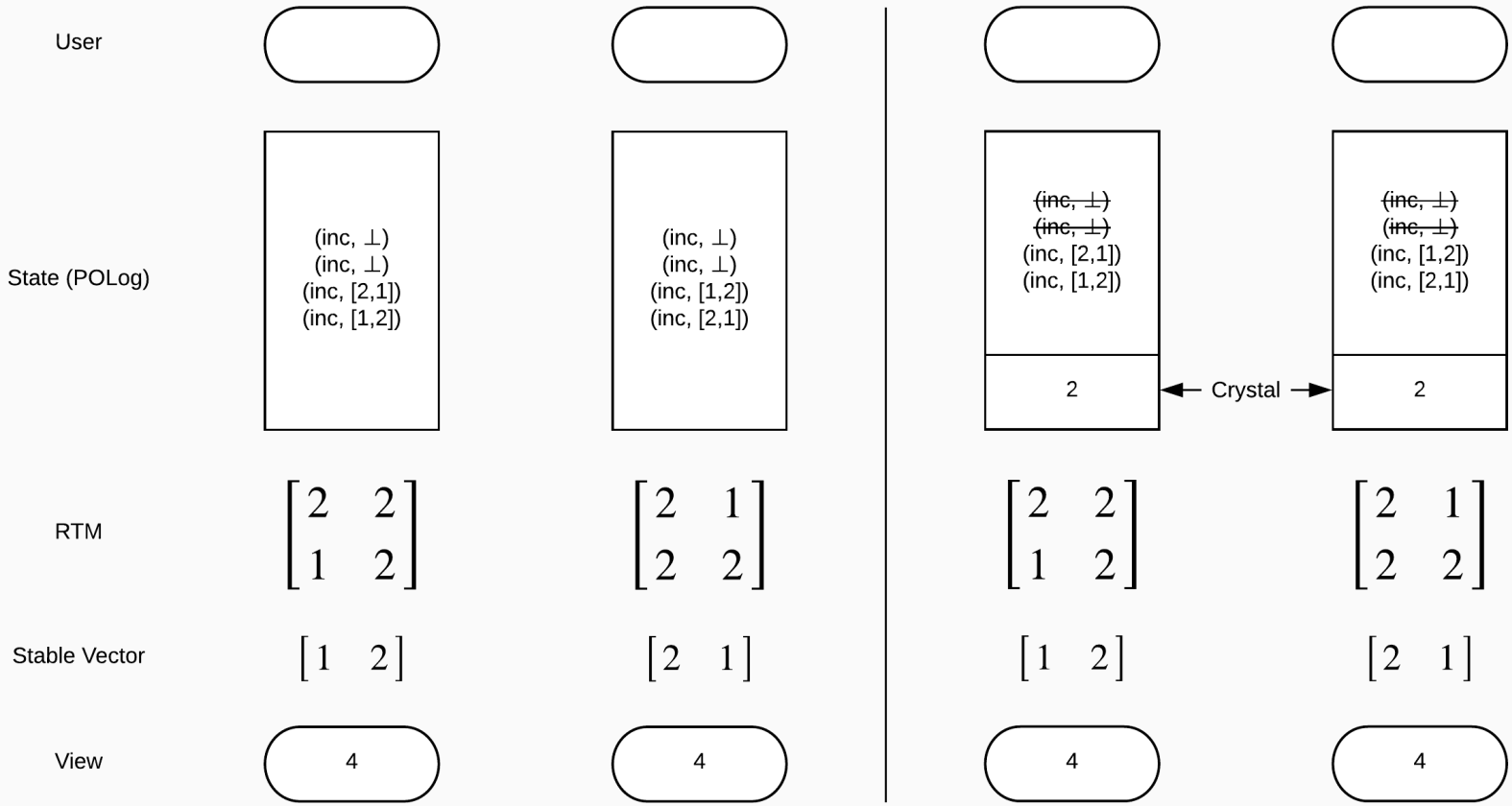


Compact Solution



Compact Solution

The implementation, more optimizations can be done. For instance, when *stabilizing* an operation, instead of keeping the *inc*/*dec* operations and replacing timestamps with $\perp$ we can do the following:



Final Remarks

Final Remarks

- In our examples (and specs) we consider the more intuitive semantics for the *reset*: a *reset* operation cancels all operations in its causal past, without affecting concurrent operations.
  - Nevertheless, it is possible to support an alternative reset semantics, in which a reset also cancels concurrent operations, with some simple modifications
- To be able to apply causal stability, making a POLog a multiset was an essential ingredient
- Also, we used the GCounter in our examples (only *inc* operations) for the sake of simplicity
  - it is possible to build any counter (e.g. PN-Counter) allowing *dec* operations as well

Questions?

bit.ly/compact-reset