How to Write a Reliable Collaborative Activity Without Even Trying

An optimistic look at what we can do to make writing collaborative software easy. Really easy.

Ben Schwartz
Volunteer, Harvard

What is collaboration?

Unified abstract state.

“If two people are working on two different documents, they're not collaborating.”

Story: Buggy Sharing

What does Telepathy guarantee?

Integrity

Causality

What does it not guarantee?

Nonexistence of parallel universes

Message delivery across split+merge

(Netsplit!)

StopWatch

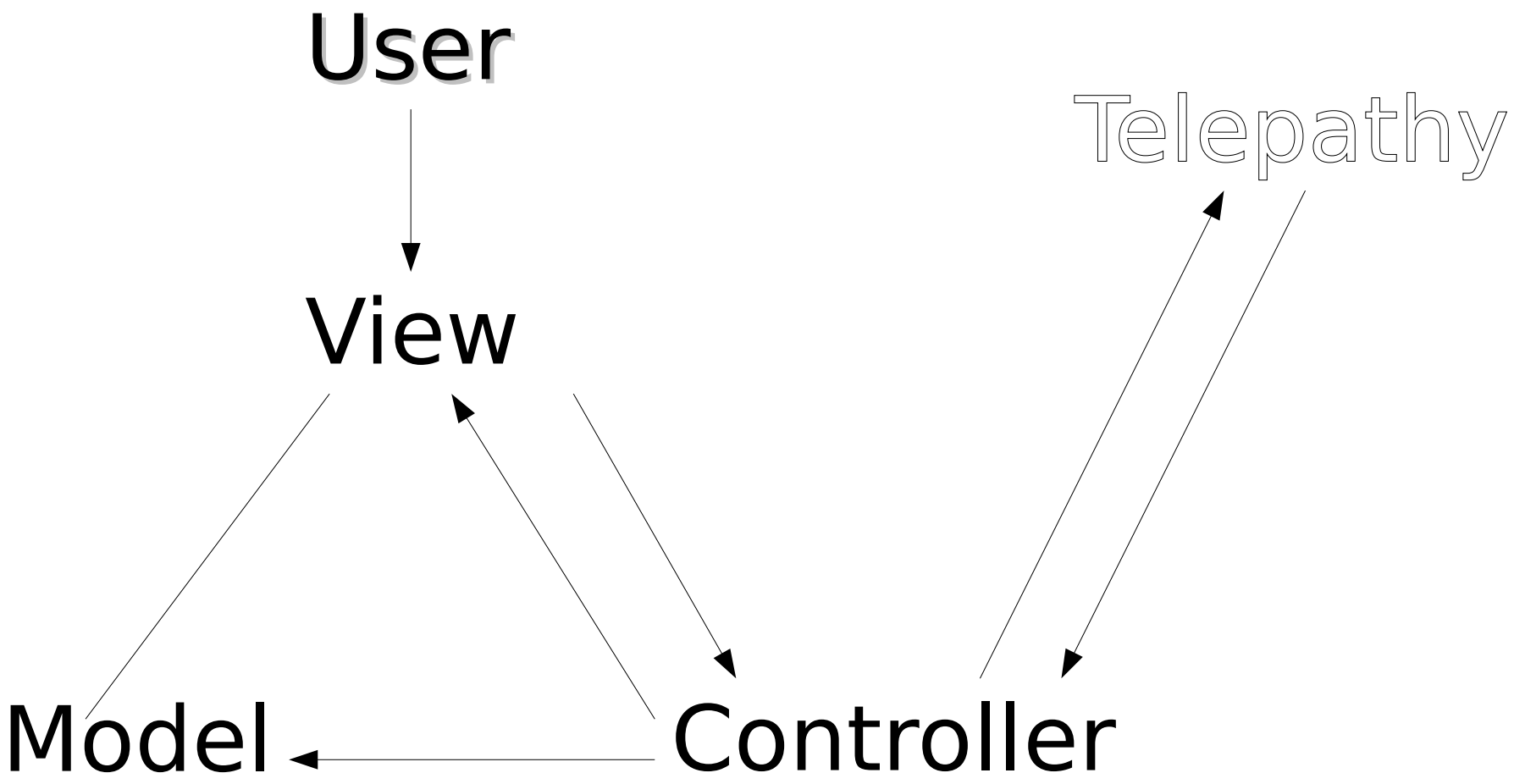
Every button click generates an Event.
e.g. (“Pause”, 1207194405.190577072)

Each stopwatch is a state machine over Events.

The order in which Events are received *does not matter*.

When users join or merge, they get sent the history.

StopWatch-1 Control Flow



It works!

but hey, who put this D-Bus call in my perfectly good program logic! Now it's all ugly.

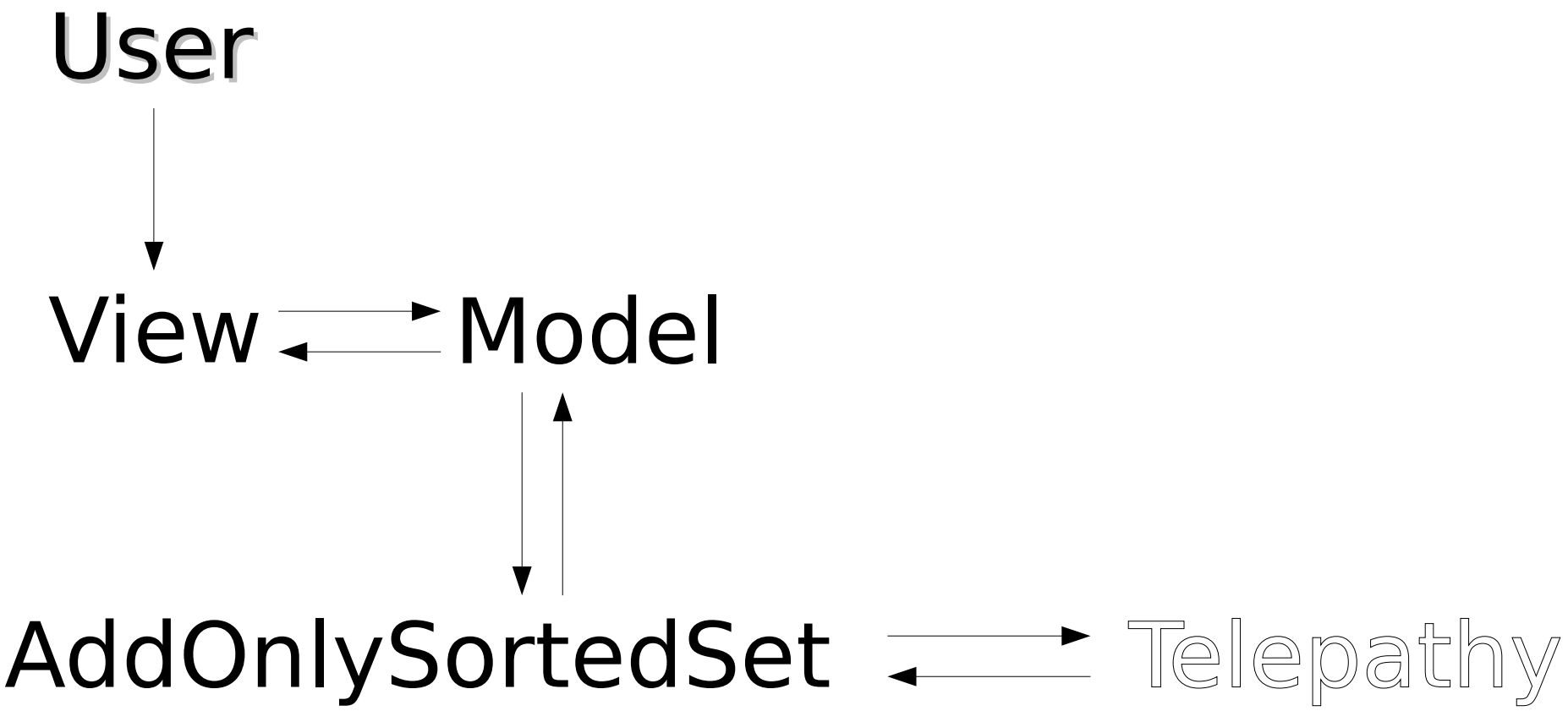
Yuck.

Hard to write. Not reusable.

Problem: You hate networking code.

Solution: Put it in a library.

StopWatch-3 Control Flow



Distributed D-Bus Objects

Original idea: 3 Kinds of DObjects

1. Unordered. No ordering.
   Archetype: AddOnlySet

2. Causal. There is a total order, but messages may arrive out of order.
   Archetype: Distributed Dict

3. Strongly Ordered: Global lock to enforce ordering.
   Archetype: Card Game

Realization #1

CausalObject can be implemented as a layer over UnorderedObject.

Strongly Ordered is *hard* (see PAXOS).

You can do almost anything with Unordered and Causal!

dobject.py

Provides: HighScore, AddOnlySet, AddOnlySortedSet, CausalDict, Latest

(coming soon: DstringIO, a new name)

Realization #2

D-Object is perfect for offline collaboration:

No merge ever fails.
Shared copies never diverge.

Vision

Activities store state in objects provided by DObject. DObject handles the rest:

tracking Joins and Quits
syncing with an upstream server
serializing to the datastore
providing an Undo GUI

Criticism: Efficiency

Space Efficiency:
Naïve DObjects might keep every message.
Solution: Keep only what is necessary to interpret future messages (already done).

Network Efficiency:
Current implementation broadcasts too much. O(N) join, O(N^2) “unsplit”. Better behavior is possible.

Criticism: Power

“My collaboration can't be expressed as a set of independent messages.”

Perhaps.

(If you don't like it, you don't have to use it.)

dobject.py Unordered API

handler.send(msg): broadcast a message

receive_message(msg): accepts and processes a message sent via handler.send().

get_history(): returns an encoded copy of all non-obsolete state

add_history(state): accept and process the state object returned by get_history()

dobject.py Causal API

Exactly the same, except:
handler.send(msg) returns an index for the message

receive_message(msg, index)

The handler generates each index as a tuple: (monotonically increasing int, random 64-bit tiebreaker)

CausalDict: Interface

Three components:
1. Complete standard python dict interface
2. CausalObject interface
    - Requires that CausalDict constructor take a CausalHandler
3. Callbacks
    - When someone else makes a change, the callback is triggered.

CausalDict: Implementation

2 Messages:
(ADD, key, value), index
(DELETE, key), index

ADD:
if not key in ind or ind[key] < index:
    d[key] = value; ind[key] = index

DELETE:
if not key in ind or ind[key] < index:
    del d[key]; ind[key] = index

DELETE works even if it arrives before ADD

CausalDict: Implementation

CausalDict.get_history() returns a serialization including both d and ind.

CausalDict can forget deleted values, but not deleted keys.