Responsive Data: Initial Design

tjdwill

April 25, 2024

Abstract

This document details the initial design of what I call *responsive* or *active* data. The idea is to have self-aware data than can respond to inquiries in pursuit of an idealized data container with $\mathbf{O}(1)$ access and sort time.

1 Introduction

Current data containers in programming contexts are passive, meaning in order to access specific data, one (traditionally) iterates through the container. However, as the container scales in size, the access time may take longer depending on the location of the data element within the container.

An alternate container type would be active. Imagine a conference room filled with thousands of audience members, each with a unique card. If I wanted to retrieve a specific pattern, looking through each audience member's card would be incredibly inefficient. Instead, the natural course of action is to call out the desired pattern and have the specific audience member bring the card forward. So it is with the proposed data container herein. In this document, I propose a (very *very* early) design for an active data container and associated structures. The current name for this data structure is `ActiveArray`, though it may change in the future.

2 Infrastructure

In order to represent this concept programmatically, three classes were created: `ActiveData`, `ActiveArray`, and `NoticeBoard`. The first represents responsive, self-aware data elements. The second is a container that holds the active data elements and serve as an interface to access the data. Finally, the third class provides a method to pass queries along to the data elements for evaluation. The three structures were programmed using the `Hy` programming language, a Lisp implementation in Python that allows for Lisp-style programming with Python modules and packages[1]. As shown in Figure 1, the developed structures interact as follows. First, a user inputs a query to the `ActiveArray` object that is linked to many `ActiveData` objects. Next, the array object determines the command to send and writes the command to the `NoticeBoard` object. The data objects each read and execute the command, writing their responses to the response mechanism. The array then takes the response mechanism, filters for relevant answers, and outputs the answer to the user.

[1] `https://github.com/hylang/hy`

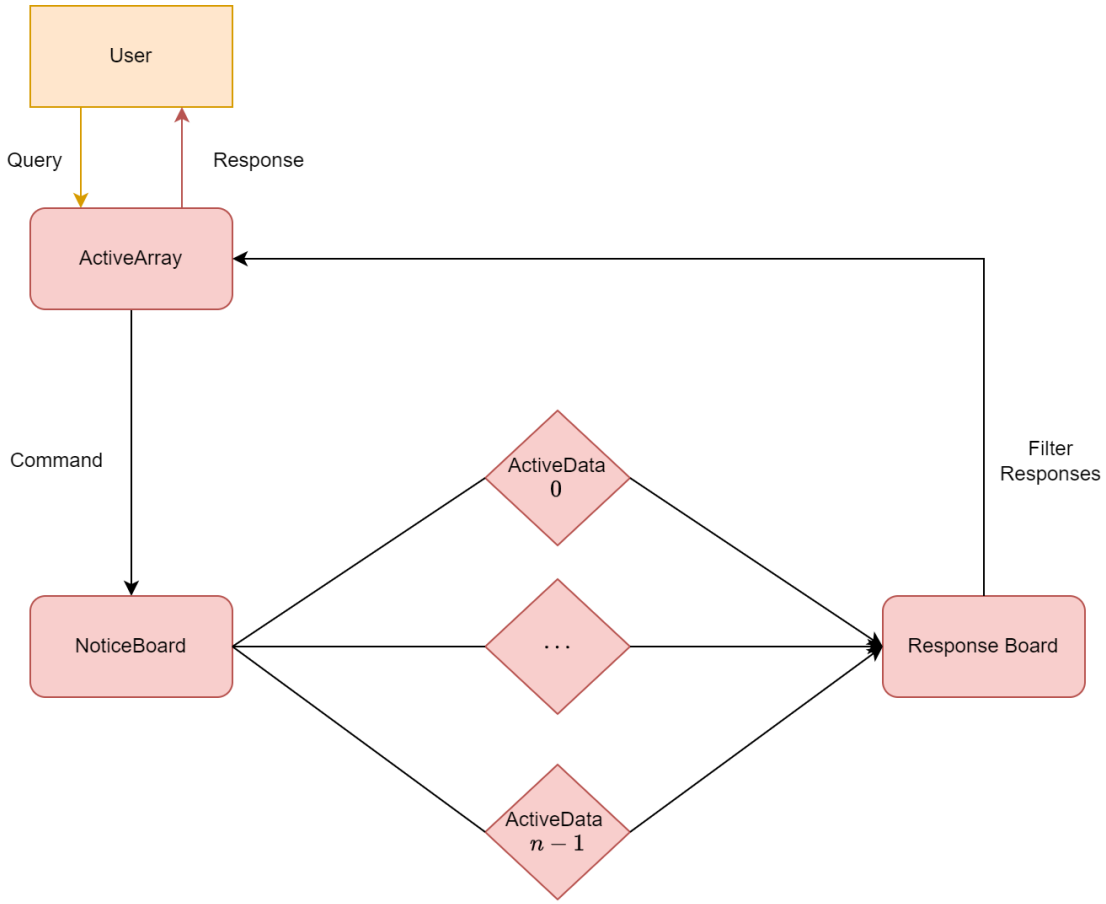

Figure 1: Structure Relationships

2.1 `NoticeBoard`

The first structure described is the `NoticeBoard`. This structure is used to pass the input commands to each data object at once. Rather, it serves as a board the array object writes to and which data objects "read" from. The (abstract) constructor is as follows:

```
NoticeBoard(author_key, canvas)
```

The `canvas` is the actual board. Currently, I use a `namedtuple` as the canvas which has fields `command` which holds the command key and `query` which is the actual Boolean function the data uses to determine if it satisfies the user inquiry. the `author_key` is a form of authentication used to ensure that the only object that can write to the `NoticeBoard` is the object that has the key used during instantiation. Specifically, in this design, the `ActiveArray` object instantiates the board by using its own memory address as the `author_key`. None of the data objects hold a reference to the array and Python promises unique object IDs while the objects persist, effectively ensuring that only the array object can update the board.
Attribute(s)

- `content`: Returns command and query

2.2 `ActiveData`

The next structure is `ActiveData`. This basically "gives life" to a data element. The main idea of this is that structure is that each object lives in its own execution thread. The object keeps track of its value and its assigned index to respond to queries it receives through the `NoticeBoard`. Unfortunately, due to each element having its own thread, having a lot of data elements results in poor performance due to context switching overhead. The Python Global Interpreter Lock doesn't help either. It is unclear to the author what programming concept/primitive can be used to provide a way for an arbitrary number of objects to remain dormant and respond to some event.

2.2.1 Constructor

In any case, let's describe the object. `ActiveData` objects are constructed with the following parameters:

```
ActiveData(
        index,
        value,
        event_flags,
        notice_board,
        response_board
)
```

- index (int): the object's position in the container.
- value (Any): the object's value
- event_flags (list): A list of threading event flags for synchronization
- notice_board (NoticeBoard): the means of getting commands
- response_board (dict): the method of responding to queries.

An `ActiveData` object's index is provided on construction during the initialization of the container object. It maintains the original order of the input container. Sorting and other index-modifying objects are not currently implemented at this time. The event flags are needed for synchronization and to prevent data races. There are currently three such flags. The first is a flag called `fl_command` used to force the object to block until a new command is available on the notice board. Second is the `fl_write` flag used to inform the container object that the individual data object has responded to the command. Finally, `fl_resume` is used to force the data object to block until the container object has received all responses.

The response board is a way of preventing data races without needing to pass around a lock. The board is a dictionary where each key is an `ActiveData` object's memory address. Initially, all values are set to `None`. Because an object has a reference to itself `self`, it is able to use its object ID to write to the dictionary, effectively giving each data object its own slot. This allows all of the objects to write to the dictionary without fear of overwriting another object's entry.

2.2.2 Methods

At the time of writing, there are five methods for this class, four of which are used to handle commands.

- `ActiveData.val-from-bool(query: Callable)`: Retrieves value if data satisfies the provided query.
- `ActiveData.val-from-idx(query: Callable)`: Special case of `ActiveData.val-from-bool`. Retrieves value if index matches.
- `ActiveData.idx-from-bool(query: Callable)`: Retrieves index if data satisfies provided query.
- `ActiveData.spin()`: Repeatedly waits for next command and executes it.
- `ActiveData._shutdown(query: Callable)`: Breaks the `ActiveData.spin` loop, allowing the object to be garbage collected.

2.3 `ActiveArray`

The final structure defined in this first iteration is `ActiveArray`. This class defines a container—or rather, a manager—for the `ActiveData` objects. It is through this object that a user interfaces with the contained data. It's also responsible for instantiating the `NoticeBoard` and the collection of `ActiveData`.

2.3.1 Constructor

```
ActiveArray(data)
```

`data` in this case is a list or tuple of data elements (preferably homogeneously typed, but this is not enforced). On initialization, each data element is converted into an `ActiveData` object whose `spin` method is called in a new thread. The new active objects' ids are then used to instantiate the response board (the dictionary).

2.3.2 General Methods

- `ActiveArray._send-command(command: int, query: Callable)`:
  Publishes command to the `NoticeBoard`.
- `ActiveArray._send-response()`: Output the coalesced responses from the data objects
- `ActiveArray._shutdown-data()`: send the shutdown signal to all `ActiveData` objects managed by this container.
- `ActiveArray._val-from-bool(query: Callable)`: `_send-command` alias
  to call `ActiveData.val-from-bool`.
- `ActiveArray._val-from-idx(index: int)`: `_send-command` alias
  to call `ActiveData.val-from-idx`.
- `ActiveArray.where(query: Callable)`: Public `_send-command` alias
  to call `ActiveData.idx-from-bool`.

2.3.3 Special Methods

Besides the expected `__repr__` and `__str__` implementations, `ActiveArray` implements `__len__`, `__getitem__`, `__enter__`, `__exit__`, and `__del__` for ease of use. The `__del__` method calls the shutdown method to exit all of the data threads before the array is garbage collected, thereby preventing hanging threads. However, it's better to use the object within a context manager to ensure this happens even in case of error. `__getitem__` is implemented such that user can provide either an integer (negative index values are supported) or a lambda function that returns a boolean value.

3 Example(s)

Here, a few examples are provided to show how to use the developed work. They are written in the `Hy` language. I deliberately forgo a context manager for the command-line examples to get the output piece-by-piece, but it is better to use one.

```
>>> (import
        threading
        activedata [ActiveArray]
        numpy :as np)
>>> (setv data (
        lfor _ (range 3)
                (np.random.randint 0 50 :size #(2 2))))
>>> (print #* data :sep "\n\n")
[[39 33]
[25 24]]

[[32 30]
[42 22]]

[[25  7]
[29 19]]
>>> (threading.active_count)
1
>>> (setv actv_arr (ActiveArray data))
>>> (threading.active_count)
4

>>> ; Get all arrays whose entries are all even.

>>> (get actv_arr (fn [x]
        (. (np.equal (% x 2) 0) (all))))
[array([[32, 30],
[42, 22]])]

>>> ; Get all arrays that have at least one even entry

>>> (get actv_arr (fn [x]
        (. (np.equal (% x 2) 0) (any))))
[array([[39, 33],
[25, 24]]) array([[32, 30],
[42, 22]])]
>>> (setv a (get actv_arr 1))
>>> (setv b (get actv_arr (fn [x]
        (. (np.equal (% x 2) 0) (all)))))
>>> (= a b)
True

>>> ; Get all arrays that sum to less than 100.
```

```
>>> (get actv_arr (fn [x] (< (x.sum) 100)))
[array([[25,  7],
[29, 19]])]
>>> (del actv_arr)
>>> (threading.active_count)
1
```