# MultiChor Demo Exercise

## Instructions

Pull the MultiChor repository:

```
git clone -b auction-demo git@github.com:ShapeOfMatter/MultiChor.git
```

Enter its dir and check that you're at the HEAD of the `auction-demo` branch.

Run the (reduced) unit tests to confirm you're set up.

```
cabal test -f test
```

This will also build the whole project, so it may take a little while the first time. One more likely source of problems is your GHC version; version 9.10.1 is preferred.

Open `examples/Auction.hs`. Observe that there's an example choreography `auction` on line 33. You'll be editing this to correctly implement the below protocol.

The goal here is to find specific short-comings of MultiChor as a library that people might actually use. The goal is *not* to test your own skill, or acquire a perfect implementation of the below protocol. Set yourself a timer for two hours, and quit when it goes off. Ask Mako questions, including about how to do particular things, at any point.

## Exercise

A group of companies are setting up an automated system that will run at midnight every night to set the price of doodads for the following day. Five of the companies are buyers, and there is one seller. There is also a "proctor" participant, who provides some oversight. This will be a semi-blind Vickrey auction: Each of the buyers will send their bids to the seller, who will inform everyone of the bid amount and identities of the top two bids. Because the parties are all well-informed and doodads don't change much in value from one day to the next, ties are likely. In the event of a tie, preference will be given to the various parties randomly. The parties trust each other; multi-round commitments are considered unnecessary.

Here is the specific protocol:

1. All buyers send their bids to the seller and the proctor.
2. **IF AND ONLY IF** there is a (possibly many-way) tie for highest bid:
    - The proctor randomly chooses one of the highest-bidding buyers and sends that choice to the seller.
3. The seller sends everyone the two tuples (winner, bid) and (second-place, bid). In the case of a many-way tie, it doesn't matter who is chosen as second-place.
4. All parties print the name of the winner and the amount of the second-place bid.

## *Nota bene*

- The Hackage documentation may be easier to navigate than the source code in `src/`, but it may be a little out of date. In particular, we used to use the word "enclave" instead of "conclave".
- If you need a `KnownSymbol` or `KnownSymbols` constraint, you can usually just add it wherever you need it.
- If you need to work with a `Quire`, notice that its instances for `Functor` *etc* don't give you access to the party names; you can use `stackLeaves` to make a new `Quire` and `toLocTm` to get the term-level name of a party.
- The monad `CLI IO` is a shim around `IO`. It satisfies `MonadIO` (so you can always use `liftIO`), and it affords `getInput` and `putOutput`, which are like normal input and output except they can be mocked during testing.
- Most of the test cases have been de-activated so you can run the tests quickly and often. Should you want to turn them all back on, change `examples/Tests.hs` line 41.
- Effectively calling `main` via `cabal run...` would require coordinating seven open shells; I don't suggest you bother.

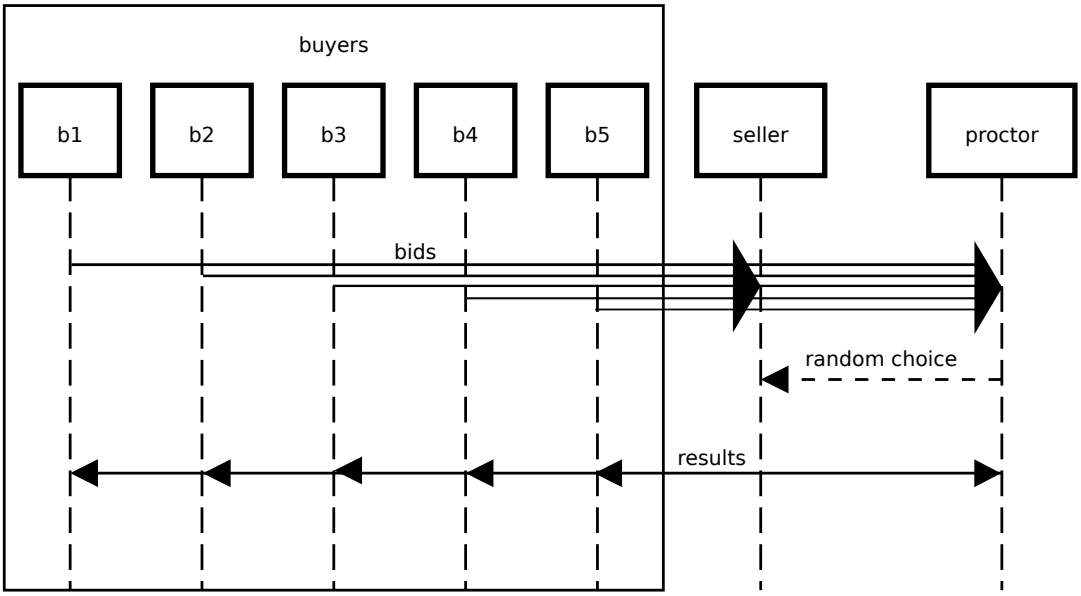


Figure 1: sequence diagram