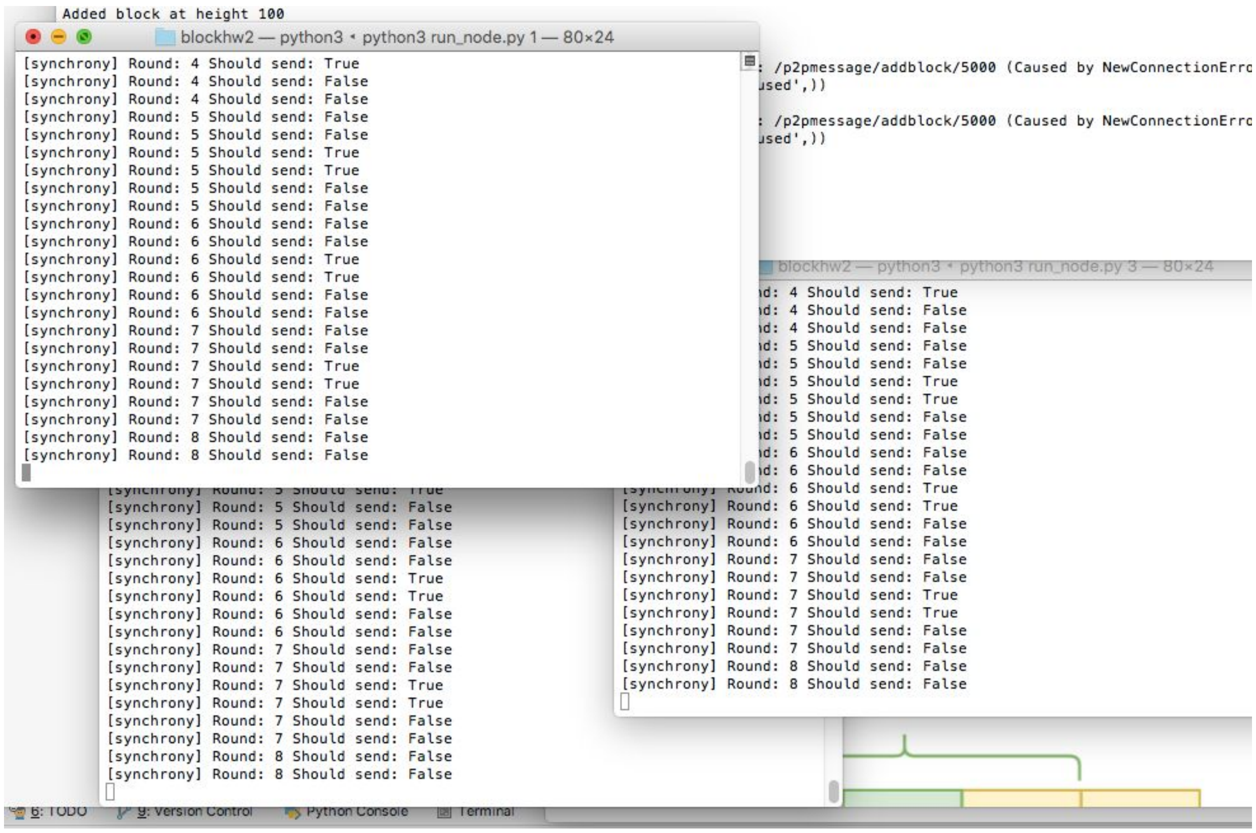
Blockchains hw2
Daniel Terry, Robert Wolfe,Yan Jiang

1.1) Done. tested.
1.2) 3 nodes, no interruptions, all were synchronized at the end. When one node was interrupted and restarted again after the pow script was complete, the interrupted node was synchronous with the others until the point of interruption, and after that, it has no data. When the node was interrupted while the pow script was running, and restarted while the pow script was still running, it had the same effect. Our current gossip protocol needs a feature for a node to join an existing chain, or to catch up if it is has been offline. Such a method would need a way to gossip all previously missed messages to the node coming back online.
1.3) A hardcoded list of participants in a p2p network is highly inflexible. It also presupposes an existing list of “approved” nodes, meaning the blockchain is not permissionless. There should be a mechanism for nodes to communicate with each other about who is currently known to be active on the network, so it is possible for nodes to join and leave the network over time without impacting the performance of the overall blockchain. The closest match in the bitcoin documentation is the addr/getaddr request.
2) Coded. Mistake in testing sent to phil. Screenshot of the start_synchrony.py test. Nodes are synchronized in all rounds.


blockhw2 — python3 ◂ python3 run_node.py 1 — 80×24
blockhw2 — python3 ◂ python3 run_node.py 3 — 80×24
6: TODO
9: Version Control
Python Console
Terminal

```
Added block at height 100
```

```
[synchrony] Round: 4 Should send: True
[synchrony] Round: 4 Should send: False
[synchrony] Round: 4 Should send: False
[synchrony] Round: 5 Should send: False
[synchrony] Round: 5 Should send: False
[synchrony] Round: 5 Should send: True
[synchrony] Round: 5 Should send: True
[synchrony] Round: 5 Should send: False
[synchrony] Round: 5 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: True
[synchrony] Round: 6 Should send: True
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 8 Should send: False
[synchrony] Round: 8 Should send: False
```

```
: /p2pmessage/addblock/5000 (Caused by NewConnectionErro
used',))

: /p2pmessage/addblock/5000 (Caused by NewConnectionErro
used',))
```

```
d: 4 Should send: True
d: 4 Should send: False
d: 4 Should send: False
d: 5 Should send: False
d: 5 Should send: False
d: 5 Should send: True
d: 5 Should send: True
d: 5 Should send: False
d: 5 Should send: False
d: 6 Should send: False
d: 6 Should send: False
ound: 6 Should send: True
[synchrony] Round: 6 Should send: True
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 8 Should send: False
[synchrony] Round: 8 Should send: False
```

```
[synchrony] Round: 5 Should send: True
[synchrony] Round: 5 Should send: False
[synchrony] Round: 5 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: True
[synchrony] Round: 6 Should send: True
[synchrony] Round: 6 Should send: False
[synchrony] Round: 6 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: True
[synchrony] Round: 7 Should send: False
[synchrony] Round: 7 Should send: False
[synchrony] Round: 8 Should send: False
[synchrony] Round: 8 Should send: False
```

Problem 3). Coded.

Problem 4
One function to choose the leader, as described in the notes, is to use player (e mod n) + 1 where e is the number of the epoch. This function simply rotates through each player. It's simplicity is a plus and it guarantees exactly one leader per epoch, but having a set order for the leader makes it possible for adversaries to plan attacks. Also, there has to be a set list of players for at least the duration of the round if everyone is to agree on a rotation through the players.

Another way to choose would be to require a puzzle solution based partially on the output of the last epoch. Only messages that included such a puzzle solution would be considered valid, and the first player to multicast such a message would be the leader for the next round. Such a function would introduce randomness into the order of leaders and could function in a permissionless setting. The cons are it requires extra computation and agreement for every epoch. Some epochs could have multiple players get a solution at basically the same time. One possible solution to this is to have a timestamp and the players' public keys be part of the hash, and if multiple solutions are found within a certain time of each other, move on to the next epoch. Also, a player's likelihood to be leader would be proportional to their fraction of computation power in this protocol, which could lead to problems with chain quality and differing "liveness" times for different players.

Problem 5
5.1) (a) This protocol satisfies Validity by the same proof shown for the protocol in the notes. If the sender is honest, they will sign at most 1 message m. Because honest players will only add a message to their set if the sender has signed it, m is the only message honest players will add, and they will all add m in round 1. Thus, all honest players will output m even in a protocol that stops at round 2.

(b) Let the sender be 1, the faulty receiver be 2, and the two honest players be 3 and 4.
Round 0: 1 sends the same message m1 to 3 and 4.
Round 1: 3 and 4 both receive $m1_{pk1}$ and both add m1 to S_3 and S_4 respectively.
3 multicasts $m1_{pk1,pk3}$ and 4 multicasts $m1_{pk1,pk4}$.
2 sends $m1_{pk1,pk2}$ to 3 and some $m2_{pk1,pk2}$ to 4.
Round 2: 3 receives $m1_{pk1,pk4}$ and $m1_{pk1,pk2}$ so S_3 remains {m1}
4 receives $m1_{pk1,pk3}$ and $m2_{pk1,pk2}$ so S_4 becomes {m1, m2}
Output:
3 outputs m1 and 4 outputs 0

It the protocol were to run another round, the attacker couldn't create a faulty message with enough signatures for an honest player to accept it on the last round. If the attacker tried sending conflicting messages to an honest player before the last round, all honest players would have it on the next round and would output the same thing.

5.2) A protocol that satisfies consistency and liveness but not future self-consistency can be constructed as follows:

For every player i:

- When protocol begins, let $\mathbf{MEMPOOL_i} = \varnothing$
- Whenever i receives a transaction x from *Z* or from another player j, if x isn't already in $\mathbf{LOG_i}$ or $\mathbf{MEMPOOL_i}$, add x to $\mathbf{MEMPOOL_i}$ and broadcast x to all other players
- Whenever $\mathbf{LOG_i}$ changes, remove any $x \in \mathbf{LOG_i}$ from $\mathbf{MEMPOOL_i}$

For epoch $e \geq 1$:

- Let *leader* = (e mod n) +1
- *Leader* sets m equal to the result of adding $\mathbf{MEMPOOL_{leader}}$ to $\mathbf{LOG_{leader}}$ and shuffling the order of the transactions
- *Leader* acts as sender in a multi-value BA protocol sending out m. All other players act as receivers. This BA protocol outputs m if the sender is honest and $\perp$ otherwise
- At the end of the protocol, if the output is $\perp$, all players j (including *leader*) leave $\mathbf{LOG_j}$ unchanged. Otherwise, all players j (including *leader*) set $\mathbf{LOG_j}$ to the output of the BA protocol.

This protocol satisfies consistency and liveness but not future self-consistency because shuffling the order of transactions means an earlier **LOG** will likely not be a prefix of a later **LOG**. Such a consensus protocol does not implement a secure public ledger because without future self-consistency, the ledger would be vulnerable to double spending attacks. More generally, without a coherent history, the ledger will have no meaning and do nothing to secure any transactions. Many transactions will have different or no meanings when taken in a random order.

5.3) In a permissionless setting, it is easy for an adversarial player to generate as many fake players to control as they like. Therefore, they could overcome any bound on the fraction of adversarial players. Using proofs of work allows a BA protocol to run in a permissionless setting by requiring players to expend computational power to participate. Bounding the fraction of computing power is safer because an adversarial player would need to spend actual resources to increase their computing power, and it wouldn't be a trivial feat for a single player or group to acquire a half or a third of the computational power of the entire network.