Problem 1

1. Please see provided code (all tests pass).

2. Yes, when I examine the nodes after I run generate_example_pow_chain.py, all the nodes appear synchronized! However, when conducting the experiment of stopping and restarting one node, I have a very different result. I ran all nodes 1-6 but stopped node 1 halfway through and then restarted the node as the chain was almost generated. While nodes 2-6 all had a height of 100, node 1 only had a height of 50. At the end of the script's execution, node 1 was NOT synchronized with its fellow nodes. Node 1 appeared to have been left behind, and it hadn't received any additional POWs since going (temporarily) offline. In official Bitcoin protocol, a node is only "forgotten" if is it dormant for over 3 hours. In our system, node 1 was forgotten if dormant for just a few minutes. I would suggest that our gossip protocol be updated to match the Bitcoin protocol of giving the node a window of 3 hours to come back onto the scene and be updated. In order to ensure consistent gossip spread, a node needs to have some flexibility in going and coming offline, as temporary outages are inevitable in any system.

3. Yes, this does suggest that we are missing the flexibility to accept any actor to the network, thereby not achieving permissionless status. With the list of peers hard-coded, it is impossible for a new person to join without having to change the code in our config file. Our system is missing the ability to look for and add any new actors who come onto the scene. The closest analogous message type in the Bitcoin protocol documentation would be getaddr. According to the documentation, getaddr is a message type that searches for potential additional active nodes in the network. The message asks a node they already know to see if it knows of any other active peers and the node does, it responds with their addresses[1]. This ensures that the system is open to adding new nodes and actors without being constrained to what is hard-coded in the config file.

Problem 4

We want two main properties from a blockchain/consensus protocol: consistency and liveness. Consistency means that if we take any 2 nodes, at any given point in the execution, their LOGs need to be the same. Liveness means that if someone wants to add a transaction, it "eventually" (within some bounded time) gets added to everyone's LOG[2]. BFT protocol in turn presents different statuses of functioning / not functioning to different components in the system. This creates difficulty for the components to reach an agreement about who is functioning or not, as they are all seeing different statuses for the same other components. However, they **must** reach agreement to decide who to trust and

[1] https://en.bitcoin.it/wiki/Protocol_documentation

[2] Pass, Rafael, *Consensus for Blockchains Preliminary Draft Notes*, 3.12.2018

not trust as a sender/leader. A system is BFT if it can still function under such ambiguity and confusion and ensure consistency and liveness, despite the difficulties.

One function for choosing a leader is derived from multi-round sessions of transactions. In epoch *e*, we let player （*e* mod *n*） + 1 be the sender. As the first player in the round-robin, his job to collect all new transactions into a single block, which he then sends out to the rest of the players in the round via the BA protocol. The other players in the round then attach the output of this BA protocol to the end of their current logs. Note that when the protocol begins, MEMPOOL_i is empty. Then, whenever player *i* receives a transaction x from Z (or another player *j*), and *x* is NOT already present in player *i*'s LOG or in MEMPOOL_i , *x* is added to MEMPOOL_i and broadcasted to all other players. If player *i*'s LOG ever changes, then any transactions *x* within their LOG is removed from their MEMPOOL as well. Now, for subsequent rounds *e*, the leader is set to (*e* mod *n*) + 1, *m* is set equal to MEMPOOL_i, and the leader then acts as the sender by sending out this *m* (MEMPOOL_i) and all other players act as receivers. Then, at the very end of the BA protocol, each player *j*, including the leader of this round, sets *new_trans* to the output of the protocol, and subsequently append this new transaction to their individual LOGs. The cycle then repeats with a new leader each round. This function ensures both consistency and liveness, as all honest players will always have the same LOGs (since they always have same *new_trans* and therefore same transaction appended to their LOGs), and we know liveness is guaranteed because any player *i* adds any transaction *x* to their own LOG within a fixed polynomial time.[3]

Another function for choosing a leader would be the DoubleMod method. The DoubleMod method picks a leader in the following way: first, we take the previous block hash and calculate mod 10 on this hash. Let's also assume that nodes are indexed from 0 to 9. Taking the previous block hash and calculating mod 10 will give us a range of solutions from 0-9. We take this integer and choose a transaction from the previous block at this index. We then take the hash of this transaction, mod to the number of nodes in the round, and the final integer result is the number node that will be the leader.

To recap with example numbers, let's say we have a total of 5 nodes in the network (each at an index from 0 to 4). Let's also say previous block hash mod 10 gives us 4. Therefore, we chose the transaction at index 4 in the previous block. Once we take hash of this transaction and set it to mod 5, and the result is 3, then the node at index 3 is the leader.

The qualitative differences between these two functions is that while both use an element of randomness to pick the leader, in DoubleMod the randomness is based entirely on the results of several modulo calculations.

Problem 5

1. (Dolev-Strong) Consider the Dolev-Strong BA protocol (which you implemented above) in Figure 4.1 of the lecture notes, and consider a setting with 4 players. Use the abstract protocol provided in the notes for your solutions, not your above implementation. Recall that we have shown that this protocol is secure w.r.t. 2 faulty players as long as we run it up to round 3. What happens if we consider a variant of the Dolev-Strong protocol that stops after round 2.

---

[3] Pass, Rafael, *Consensus for Blockchains Preliminary Draft Notes*, 3.12.2018

(a) Show that this protocol still satisfies Validity.

The situation for Validity after Round 2 is similar to the situation at the end of Round 3:

If the sender is honest, it will sign at most 1 message. As such— since honest players only add a message m to their set if the sender has signed it—it follows by the unforgeability of the signature scheme that m is the only message that can be added to their set.

Additionally, all honest players will add m to their set in round 1. Therefore, when the protocol ends after Round 2, m will be the only message in their set and they will all output m.

(b) Show that this protocol does not satisfy Consistency w.r.t. 2 faulty players. (Hint: Consider a situation where the sender and one of the receivers is faulty.) Explain what prevents your attack if we run the protocol for one more round.

Let the two faulty players be denoted as P1 and P2, where P1 is the sender and P2 is a receiver.

Let the two honest players be denoted as P3 and P4, where both of them are receivers.

These players have public keys pk1, pk2, pk3, pk4, and secret keys sk1, sk2, sk3, sk4, respectively.

Round 0: Sender P1 signs and sends mpk1 to P3 and P4, and signs and sends a different message m'pk1 to the faulty P2.

Round 1: Honest player P3 and P4 add m to their sets. However, faulty player P2 signs and sends m'pk1,pk2 to P3 but not P4.

Round 2: P3 adds m' to its set since it has 2 votes and the sender's signature, whereas P4 still only has m in its set.

From the above transaction, we can see that, at the end of Round 2, P3 will output 0 whereas P4 will output m. Therefore, Consistency is violated in this case.

2. (Future Self Consistency) Use any BA protocol (for sending strings) to construct a consensus protocol which satisfies Consistency and Liveness, but not Future Self-Consistency. Intuitively explain why your protocol does not implement a "secure public ledger".

The protocol can be constructed by modifying the protocol in Figure 5.1:

For every player i:

- When protocol begins, let MEMPOOLi = empty.
- Whenever player i receives a transaction x from Z as input, or from some other player j, if x isn't already in LOGi, or in MEMPOOLi:
    - Add x to MEMPOOLi;
    - Broadcast x to all other players.
- Whenever a player i's LOGi changes, remove any transaction in LOGi from MEMPOOLi

For epoch e >= 1:

- Let leader = (e mod n) + 1;
- Let m = MEMPOOLi at the beginning of the epoch;
- Leader acts as the sender in a multi-value BA protocol sending out m. All other players act as receivers.
- At the end of the BA protocol,
    - If the LOGj for player j is empty, then each player j (including the leader), lets *new_trans* be their output of the BA protocol, and then appends *new_trans* to the end of LOGj **twice**.
    - If the LOGj for player j is *not* empty, then each player j (including the leader), lets *new_trans* be their output of the BA protocol, **delete** the last **one** transaction in LOGj, and then appends *new_trans* to the end of LOGj **twice**.

Since the mechanism and timing for adding new transactions in this protocol are exactly the same as the original protocol in Figure 5.1, the Liveness and Consistency properties of this modified protocol should automatically follow. However, since the protocol would remove one duplicate version of the previous transaction from LOGi every time it needs to add a new (and a duplicate) transaction, future self-consistency would be violated when the transaction with a duplicate copy in the LOGi changes after each epoch. For example, based on this protocol, if the first transaction is represented as A, the second transaction is represented as B, and the third transaction is represented as C, then an honest player's LOG after the first three epochs would look like:

LOG after Epoch 0: *empty*
LOG after Epoch 1: AA
LOG after Epoch 2: ABB
LOG after Epoch 3: ABCC

And clearly the LOG after Epoch 1 is not a prefix of the LOG after Epoch 2 or 3, and therefore the Future Self Consistency property does not hold. This protocol does not create a secure public ledger because it has no mechanism to ensure participants have deleted and added the correct number and version of transactions each time. Therefore, it is possible for an adversary to manipulate and create inconsistent transaction histories by adding just one copy of the new transaction to the LOG each time, rather than two duplicate copies as stipulated in the protocol above.

3. (Permissionless BA) Explain informally why we need to use proofs of work (and bound the fraction of adversarial computing power, as opposed to simply giving a bound on the fraction of adversarial players) to get a BA protocol in the permissionless setting. (We are not expecting a formal proof, just an intuition.)

In the permissionless setting, anyone can join or leave the protocol execution without any form of permission or authentication, and any participant may join or leave at will. This means there would be no known upper bound on both the total number of nodes in the system at a point in time and the amount of faulty participants. Since it's possible (and easy) for a malicious user to create multiple online identities, a certain type of barrier-to-entry needs to be established to prevent such malicious users from effortlessly introducing a large number of faulty participants and manipulate the consensus process in a Sybil attack. Proof-of-work (PoW) can be used to establish such a barrier-to-entry by requiring nodes to prove they have expended significant amount of energy towards solving a hard cryptographic puzzle. Such a barrier-to-entry would make it become much more expensive for a malicious user to take control of a significant number of participants in the system and therefore greatly reduce the chance of success for a Sybil attack. However, expensive still does not mean impossible. Therefore we still need to assume the computational power that the adversary can gather is bounded and will not be sufficient to create a large enough group of faulty participants to manipulate the consensus process.