CS 7610: HW-3

Name: Sumeet Mahendra Gajjar

Solution 1.a:

The Ben-or paper describes the algorithm for Byzantine resilience. Let t be the number of faulty processes, $\mathcal{N}$ be the total number of participating processes and n be the minimum number of participants required for the algorithm to succeed, then $n > 5t$ condition should be satisfied for the algorithm to succeed and be Byzantine resilient.

The non-byzantine algorithm should be modified in the following way to make it byzantine resilience:

- During the suggestion phase, in "step 2.a", a process sends the D-message $(2, r, v, D)$ if and only if the number of $(1, r, v)$ messages received is greater than $\frac{(\mathcal{N}+t)}{2}$ instead of $\frac{\mathcal{N}}{2}$.
- During the decision phase,
  - in "step 3.a", a process changes its x_p value to v if and only if the number of D-messages $(2, r, v, D)$ received is at least $t + 1$.
  - in "step 3.b", a process decides the value as v if and only if it the number of D-messages is greater than $\frac{(\mathcal{N}+t)}{2}$.

The algorithm assumes a process can determine the originator of a message that it has received and none of the messages are lost, if a process is alive long enough, it will receive all the messages eventually. The complete Byzantine resilience algorithm is as follows:

- For a process $\mathcal{P}$, let the initial value be x_p
- **step 0:** set $r := 1$
- **step 1:** Send the message $(1, r, x_p)$ to all the processes
- **step 2:** Wait till messages of type $(1, r, *)$ are received from $\mathcal{N} - t$ processes.
  - **a: If** more than $\frac{(\mathcal{N}+t)}{2}$ messages have the same value v, then send the message $(2, r, v, \mathcal{D})$ to all processes.
  - **b: Else** send the message $(2, r, ?)$ to all processes.
- **step 3:** Wait till messages of type $(2, r, *)$ arrive from $\mathcal{N} - t$ processes.
  - **a: If** there are at least $t + 1$ D-messages $(2, r, v, D)$ then set $x_p := v$.
  - **b: If** there are more than $\frac{(\mathcal{N}+t)}{2}$ D-messages then decide v.
  - **c: Else** set $x_p = 1$ or 0 each with probability 0.5.
- **step 4:** Set $r := r + 1$ and go to step 1.

Solution 1.b:

As we have seen earlier in the lecture, in order to deal with t Byzantine failures, the number of honest processes n should be greater than $3t$ i.e. $n > 3t$. We will call it the Byzantine Impossibility result. Ben-or's Byzantine algorithm assumes $n > 5t$. We also assume the total number of process is odd.

Consider a scenario where all honest processes suggests the same value x_p, and all the faulty processes suggests the opposite value. The faulty processes suggestion cannot go ahead since they do not have the majority, thus the value suggested by honest processes will be carried to the next round.

The faulty processes tries to block this progress by proposing the exact opposite value of the honest processes in every round. This is a contradiction since $n > 5t$ ensures the majority is held by honest processes. And since none of the message is lost, it can just be delayed but with a guarantee that it will eventually reach the process, it ensures the liveness of the algorithm.

In step 2.a, each process waits for $\frac{(\mathcal{N}+t)}{2}$ messages, and in this case $\mathcal{N} > 5t$, thus $\frac{5t+1+t}{2}$ which is $\frac{6t+1}{2}$ and since number of processes should be a valid positive integer it results into $3t+1$. From this and the Byzantine Impossibility result, we can say that this step is safe and thus the value v decided by the process is a result of suggestion from a honest process.

In step 3.a, each process waits for at least $t+1$ D-messages $(2, r, v, D)$ before setting their value to v. The value couldn't be suggested by the faulty processes since it would contradict step 2.a. Step 2.a ensures the value v is only suggested if agreed by a certain majority greater than $3t+1$. And since the majority of the processes are honest this value couldn't have been suggested by a faulty process. Thus if a process receive more than $t+1$ D-messages with value v it is safe to switch to that value, since there are only t faulty processes and thus 1 honest process is agreeing with the value v since that honest process received more than $3t$ messages of type $(1, r, v)$ in step 2.a.

In step 3.b, a check of greater than $3t+1$ similar to step 2.a ensures the value which is decided is because of the honest majority's suggestion.

Consider the case where honest processes have different initial value for x_p, since there are $5t$ honest processes, step 2.a ensures only the honest processes suggestion is passed to the next round. If the condition is not met in step 2.a, the processes will send $(1, r, ?)$ message which will cause the honest processes to update the value of x_p or not in step 3.c and thus changing the split causing the step 2.a to eventually succeed in future rounds.

Thus the faulty processes can always the delay the consensus but can never block the progress or hinder with the liveness of the algorithm.

The Byzantine Impossibility results defines a lower bound of $3t$ on number of honest processes. The Ben-or Byzantine Algorithm defines a lower bound of $5t$ for it to work making it is a sub-optimal algorithm.

Solution 2.a:

The main idea behind bitcoin is decentralization, there is no single authority which controls the currency. Thus a single authority cannot accept or deny a currency transaction and hence a peer-to-peer network is used. A currency transaction in context of Bitcoin can either be a create transaction where new currency i.e. Bitcoin is created or minted or a transfer transaction where certain amount of Bitcoin is transferred from one party to another.

Bitcoin uses a append-only ledger to record transactions. This ledger is publicly available and anyone in the network can verify whether a transaction is present in the ledger or not.

Now the question arises on whether a transaction is valid and should it be recorded in the ledger or not. Since it is a peer-to-peer network traditional consensus algorithms like RAFT, Paxos, etc cannot be used. Additionally, these algorithms do not provide Byzantine resilience hence another reason why they cannot be used here.

Bitcoin uses a computational puzzles along with hashes of transactions to securely record transactions in the public ledger. The ledger is nothing but a chain of blocks. Each block contains a pointer to its previous block. Along with the pointer, a block also contains a set of valid transactions. Whenever a new block is added to the ledger, a certain reward in terms of Bitcoin is awarded to the peer which adds the block to the ledger. This ledger is not global, each peer maintains a local copy of this ledger.

Let's assume for the sake of simplicity, there is a global ledger $\mathcal{L}$ and each block contains just one transaction. A new transaction $\mathcal{T}$ is to be recorded on the ledger, out of $\mathcal{N}$ peers, a peer $\mathcal{P}$ is randomly selected to add $\mathcal{T}$ to $\mathcal{L}$. Thus $\mathcal{P}$ is rewarded with Bitcoins.

This scheme of selecting random peer to record transactions is not unfair since like real life one needs to work to earn currency. Thus $\mathcal{P}$ is making money without doing any work. Thus a peer needs to work to earn Bitcoins.

This idea of random selection can be approximated by selecting peers in proportion to computing power a peer dedicates to the network and with a major assumption that the computing power can monopolize. The peers are competing against each other by using the computing power. This computing is required to solve hash puzzles of finding *nonce* and this how Bitcoin achieves proof of work.

Now, in order for a peer to add a block to the ledger, it is required to find a number *nonce*, such that when concatenated with the previous block hash and the set of transactions in the block, the hash of this string falls into a target space that is quite small in relation to the much larger output space of the hash function i.e. $H(nonce \| prev\_hash \| tx_1 \| tx_2 \| .. \| tx_n) < target$. On success, it broadcasts the block to all the peers and moves on to find or mine the *nonce* for the next block. On receiving a block, each peer verifies the correctness of *nonce* by simply running the computationally cheap hash function. Once a peer receives the new block, it stops mining for the same block and moves on to compute the *nonce* for next block.

The peer who computes the *nonce* first, wins the reward. Thus the only way to speed up the brute-force approach is by dedicating more computing power, and thus by doing more work. Thus the proof of work is task of finding the *nonce* for the block which the peer is trying to add.

The number *nonce*, can only be computed using a brute-force approach, unless a peer has a access to a quantum computer which runs Shor's algorithm and the peer knows how to make a measurement without disturbing the quantum state (:P)

The proof of work solves many problems some of which are as follows:

- It solves the problem of determining representation in majority decision making. It is essentially one CPU one vote. The majority decision is represented by the longest chain, which has the greatest proof of work effort invested in it.
- An attacker can easily modify the transactions in the previous blocks and publish new blocks.
- By using the random selection, we are taking a leap of faith that the peer is honest.
- Rewards are given just for participation which might make the system unstable since everyone wants to run a Bitcoin node for free rewards.

Solution 2.b:

Theoretical Aspect: Speaking from strict theoretical perspective, I believe one can never say that a transaction is committed. Let's consider a scenario where a fork resulted into two ledgers $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ contains a transaction $\mathcal{T}$ and $\mathcal{B}$ does not. A new block is added to $\mathcal{B}$ which makes it the longest chain. Thus all transactions $\mathcal{A} - \mathcal{B}$ are queued for commit. Just before $\mathcal{T}$ is about to commit, a fork happens. $\mathcal{C}$ and $\mathcal{D}$ are the new ledgers. $\mathcal{T}$ is added to $\mathcal{C}$ and not $\mathcal{D}$. A new block is added to $\mathcal{D}$ making it the longest chain and all transactions $\mathcal{C} - \mathcal{D}$ are queued for commit. This cycle can continue leaving a transaction $\mathcal{T}$ uncommitted forever.

Practical Aspect: As stated above, theoretically, is no verifiable commitment of the system that a block and thus a transaction will persist. However for practical usage, a transaction is confirmed or committed when it is buried deep enough in the chain. This is because probability of successful fork-attack decreases exponentially with the deepness of a transaction inside the chain, thus people rely on probabilities to gain confidence in a transaction. For e.g. If an attacker controls 10% of the network's hashing power, the attack will succeed with a probability of 0.02428 for a deepness of size 6. It also reduces the probability of the classic double-spending attack.

Deep enough is defined by the people involved in the transaction. The classic Bitcoin client will show a transaction as not committed until the transaction is 6 blocks deep. However different people can choose to use any arbitrary positive integer.

When the risk of loss due to double spending is nominal, considering inexpensive items, people may choose not to wait for a transaction to be committed, and complete the exchange as soon as it is seen on the network. When the value of item for exchange is comparable to the block reward it is recommended to wait for a transaction deepness of about 144 block (1 day) before considering a transaction committed.

One also choose to use a honesty score given to the parties involved in the transaction to determine the deepness where scores are inversely proportional to the deepness. However this approach has a trade-off of losing anonymity.

Solution 2.c:

ByzCoin - A Byzantine consensus protocol ensures to commit Bitcoin transactions irreversibly, ensuring if a transaction appears in the ledger, it cannot be removed. It is based on Practical Byzantine Fault Tolerance (PBFT) algorithm.

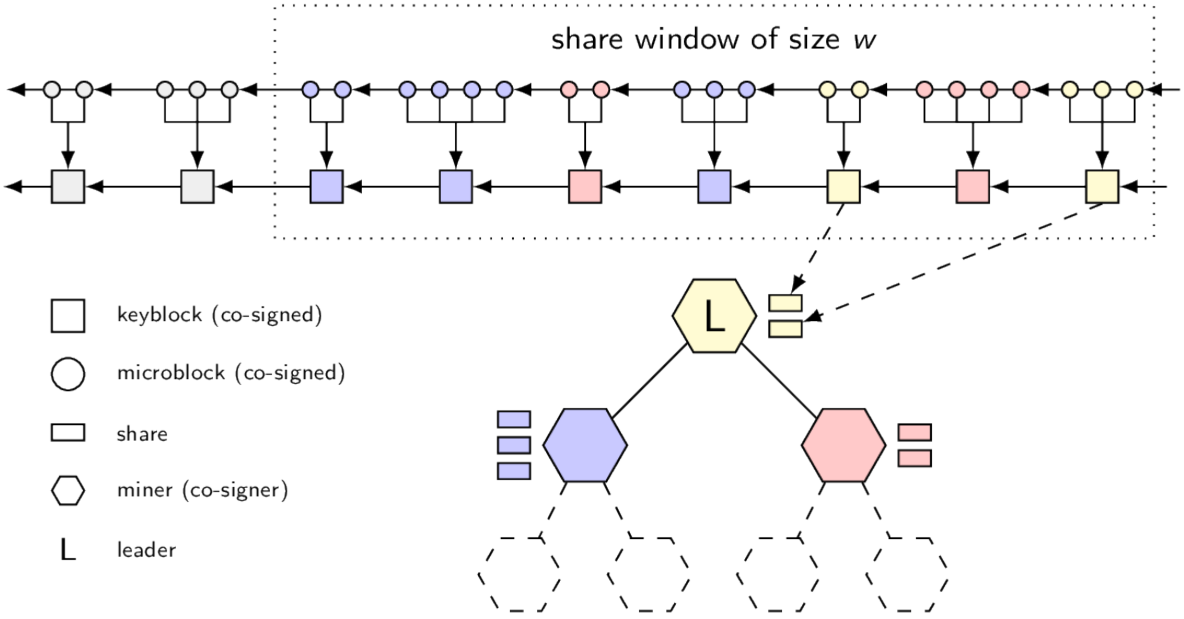

Figure 1: Byzcoin design

The lower part of Figure.1 shows the consensus group. This is a group of $n = 3f + 1$ PBFT replicas called trustees. It can have at most f faulty trustees. The group comprises of miners who have recently mined a new block within the window size. A window size w is the latest blocks in the blockchain. The miners of the latest w blocks are the members of the consensus group. A miner is rewarded a consensus group share on mining a new block, this share proves the miner's membership in the consensus group. Old shares beyond the current window expire and become useless for purposes of consensus group membership.

These trustees collectively maintain a Bitcoin-like blockchain, collecting transactions from clients and appending them via new blocks. This guarantees that only one blockchain history ever exists and that it is never be rolled back or rewritten. Thus a transaction once committed to the chain can never be removed.

Byzcoin also uses CoSi instead of MAC-based authentication to verify signatures in a constant time complexity. The details of CoSi are beyond the scope of this question. Using CoSi, Byzcoin overcomes the scalability trade-off of PBFTCoin.

Similar to PBFT, one of these trustees is the leader, who proposes transactions and drives the consensus process. The leader groups transactions into a block and initiates the commit process using CoSi. This action has to be approved by the $\frac{2}{3}$ majority of the consensus group members which guarantee Byzantine Resilience. If the leader is identifies as a Byzantine, then the consensus group initiates a view change by starting a vote to elect a new leader, the outcome of the election is valid only if $\frac{2}{3}$ majority agrees.

The upper half of Figure.1 shows that the blockchain is divided into Keyblocks chain and Microblocks chain. This solves the problem of membership for the consensus group and at the same time achieves scalability along with transaction irreversibility.

- **Keyblock:** Keyblocks are used to manage trustees consensus group membership. It forms a regular blockchain and a block is generated using the proof of work concept of Bitcoin roughly every 10 minutes. The miner of the new block gains entry into the consensus group, if it is not a member. It also becomes the next leader. A miner's valid share count reflects its voting power in the consensus group for that window. Whenever a new block is found, the reward is split among the consensus group members according to their membership share. This ensure all miners to remain active and contribute to the progress.

- **Microblocks:** Microblocks contains transactions that are proposed by the current leader. Committing to a mircoblock do not require proof-of-work, it only requires $\frac{2}{3}$ majority agreement of the consensus group. Each microblock contains a list of transactions, a hash of the previous microblock and a hash of the leader's keyblock to identify the window of the microblock. Any misconduct from the current leader while committing to the mircoblock chain would be detected by the other group members, which in turn will trigger a view change thereby removing the malicious leader.

Solution 3.a:

Spanner support external consistency, this is one of the invariant which guarantees safety in spanner. It enable Spanner to support consistent backups, consistent MapReduce executions, and atomic schema updates, all at global scale, and even in the presence of ongoing transactions. In simple terms, if a transaction T_1 commits before another transaction T_2 starts, then T_2 will see the effect of T_1 while executing. It also implies that T_1's commit timestamp is smaller than T_2's.

This is key requirement to enforced External consistency is the TrueTime API. It is a highly available distributed clock provided to applications on all Google servers. It is implemented using GPS and Atomic Clocks. TrueTime enables applications to generate monotonically increasing timestamps: an application can compute a timestamp T that is guaranteed to be greater than any timestamp T′ if T′ finished being generated before T started being generated. The TrueTime API implementation exposes an uncertainty ϵ, if this uncertainty is larger, then spanner will slow down to ensure the External consistency property is guaranteed.

If it difficult to provide external consistency, to aid Spanner to perform certain optimizations, Spanner requires the client to provide whether a transaction is a read-only or read-write before executing it.

Spanner also depends on the Paxos Group Leader's lease disjointness invariant to ensure safety. It simply means that for each Paxos group, each Paxos Leader's leader interval is disjoint from every other leader's. This is also implemented using the TrueTime API and with a assumption that two consequent leaders will have at least one replica in common in their quorums.

Solution 3.b:

A read only transaction is performed in two phases, assign a timestamp s_{read} and execute the transaction. The detailed description of both these phases is as follows:

- Timestamp assignment phase:
  - Client, at the start of the transaction declares that it is a readonly transaction.
  - Spanner does a scope expression for the given transaction. This action summarizes all the keys that will be read by the entire transaction.
  - Spanner checks whether the scope values are served by a single or multiple PAXOS groups. The scope values of the given transaction are served by multiple PAXOS groups, three to be exact.
  - Since multiple PAXOS groups are involved, a negotiation phase is required between all of the Paxos groups that are involved in the reads to assign timestamp. Negotiating across multiple groups is a complicated option and Spanner in the current implementation avoid this.

    It chooses a simpler option, Spanner executes its reads at $s_{read} = TT.now().latest$. Due to this, $s_{read} > t_{safe}$, it may have to wait for t_{safe} to advance. Once $s_{read} < t_{safe}$, execution phase begins to execute reads as snapshot reads at s_{read}. The reads are sent to replicas that are sufficiently up to date.
- Execution Phase:
  - Spanner starts a read with timestamp s_{read} on a replica if it sufficiently up to date i.e. $s_{read} <= t_{safe}$.
  - While reading a record can contain multiple values recorded from writes at varying timestamps. Spanner would return the value which is closest to the given timestamp s_{read}.

Solution 3.c:

Consider a deployment where Spanner uses 5-nodes Paxos groups to manage each tablet. A read-write transaction involving multiple PAXOS groups is executing. The transaction manager that coordinates the the two phase commit gets disconnected from the network, while this happens, two more nodes crash in the same group.

It is not clearly mentioned in the Spanner paper what happens when the coordinator leader dies or gets disconnected. An educated guess here indicates that the on going transaction will be aborted due to timeout. The two alive nodes will also detect that the lease on the leader has expired. They may start a leader election but none of them will get majority votes and thus they will be blocked.

The number of alive nodes in the group is two. The minimum number of nodes require for a successful quorum in a 5-nodes cluster is 3. Thus no matter what modifications we make to 2PC protocol or what failure protocol, there is no way the cluster can auto-recover from this failure on its own. It would be an understatement to call this scenario a failure, it is a high priority outage. A manual intervention from an operator is must in such scenario.

Taking an action based on just two out of five nodes in the group e.g. both nodes decide to commit the transaction, one becomes a leader and tries to complete the transaction does not guarantee safety neither consistency. In fact, taking such action can make the system inconsistent there by adding fuel to the fire. A quorum is strictly necessary to deal with such failures.

Following solution can be employed in the hope that the system will recover to a consistent state:

- Manually add a new node to the PAXOS group

  Assumption: each node remembers its state before crashing by storing its state on disk.

  - Each node stores its state on disk along with replicating it's state via PAXOS.
  - As stated in the Spanner paper, the state of the transaction manager is stored in the underlying Paxos group and therefore it is replicated.
  - The two alive nodes $N - 1$ and N_2 in the group will eventually detect the leader lease has been expired.
  - The nodes will start a leader election but will not able to get majority votes.
  - A new node N_{new} should be spawned with the replicated state of the transaction manager applied to it or we restart the disconnected transaction manager which reloads its state from disk.
  - N_{new} will detect it was the leader and its lease has been expired and request lease renewal from N_1 and N_2.
  - On receiving the lease renewal request, both N_1 and N_2 returns null since they voted for themselves.
  - On receiving null, N_{new} votes for either of the node, thereby electing the leader.

  - The new leader collects the pending transaction info from all the nodes and continues with aborting and restarting them.

Employing the above solution will require the transaction to be aborted and restarted. This is because the state of the lock stable which is present only on the transaction manager is mostly volatile and not replicated via PAXOS. Thus we cannot say for sure which lock leases has been expired and which has not. Thus it continuing the same transaction is safe and can introduce undefined behavior.

The above solution might sound feasible in theory, however, it might be extremely difficult to implement.

Recently on Nov 25, 2020 Amazon faced an similar outage with their Amazon Kinesis service.