Heading out for a few days vacation and thought I'd put the things that have been hogging my brainspace the last week down so that I can filter out the bad ideas when I get back. I'm looking to identify the big holes and the things I'm just not considering. The biggest areas of clarity that I need are:

1. Does this really provide for 'proof of publication' or am I missing out on some major attack vectors?
2. If gas isn't capped do things go sideways?
3. What am I not getting about the EVM that makes this concept break?

This all derived from thinking more about the propsalState minimized implementation on current evm and deciding that using one of the EVM implementations I could do away with the virtual functions and just use EVM code while hotswapping witnessed storage when needed.

The following is a specification for a scalable side chain connected to the main ethereum network via a Validation Manager Contract called [XXXXXXX].

Why?: A few things that have been said in some of the sharding threads have caused me to think that there is a generalized solution out there where we don't really need to have defined shard numbers. The following has been spun out of that line of thinking.

[XXXXXXX] has the following consensus characteristics:

- Collators and Validators post Stake to be assigned a rotating hash can qualify them to be either a Collator or a Validator.
- Each Collation has 1 initial collator and n number of validators(proper number to be determined by math)
- Any number of collations can be proposed for a particular step of consensus but only the first to qualify will be included.
- All other collations can still finalize and become 'uncles.' Uncled collations can be shared with future collators in exchange for some reward.

[XXXXXXX] has the following execution characteristics:

- Contracts are written in solidity(or a comparable evm language)
- Gas costs are not capped on the [XXXXXXX] chain
- Collators may choose to process transactions of any size, but real world market availability may affect the inclusion of large transactions.
- Collators may choose to process transactions of any size, but real world market availability may affect the inclusion of large transactions.
- Side chain contract storage is held off chain in a set of witnesses
- Users can choose to store witnesses or not depending on their data availability needs.
- Instead of proper 'storage' we use n-layered log accumulators such that canonical state is established with a witness proof that a log in the form CONTRACT, STORAGEKEY, [ADD, DEL, NULL], STEP, NONCE, VALUE is included in the accumulator
- STORAGEKEY is a storage position for the data
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- ADD updates a variable
- DEL invalidates a previous log
- NULL claims that a path is currently null
- STEP is the step of the accumulator
- NONCE is incase we update multiple times in a collation
- STORAGEKEY is a storage position for the data

- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- ADD updates a variable
- DEL invalidates a previous log
- NULL claims that a path is currently null
- STEP is the step of the accumulator
- NONCE is incase we update multiple times in a collation
- All new logs produced in the collation are merklized to produce a root
- Users can choose to store witnesses or not depending on their data availability needs.
- Instead of proper ‘storage’ we use n-layered log accumulators such that canonical state is established with a witness proof that a log in the form CONTRACT, STORAGEKEY, [ADD, DEL, NULL], STEP, NONCE, VALUE is included in the accumulator
- STORAGEKEY is a storage position for the data
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- ADD updates a variable
- DEL invalidates a previous log
- NULL claims that a path is currently null
- STEP is the step of the accumulator
- NONCE is incase we update multiple times in a collation
- STORAGEKEY is a storage position for the data
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- This can give some odd behavior with short and long variables, also mappings. Need to get Storage key to variable name mappings added to ABI: https://github.com/ethereum/solidity/issues/3736
- ADD updates a variable
- DEL invalidates a previous log
- NULL claims that a path is currently null
- STEP is the step of the accumulator
- NONCE is incase we update multiple times in a collation
- All new logs produced in the collation are merklized to produce a root
- Main net variables can be accessed as read only in sidechain contracts since collations propose a block.
- [XXXXXXX] variables can migrate back to the main chain using proofs of unduplicatable receipts.

The network provides the following transactions:

- Publish new contracts to [XXXXXXX]
- Call Functions in [XXXXXXX] Contracts

Consensus on the transactions included in each collation are determined by a collator in a randomly selected range

proposing:

- A new collation root - signed to keep it secret.
- This will be revealed in the commit operation
- This will be revealed in the commit operation
- A hash for an operation list that can be retrieved from IPFS
- A list of short term hash swaps.
- These can be used to swap out updated variables whose values have changed since transactions were published.
- These can be used to swap out updated variables whose values have changed since transactions were published.
- A set of nonce-hashed roots.
- Used by validators to claim agreement
- Used by validators to claim agreement
- An EVM blocknumber used for calculation

The collation is confirmed by a set of randomly selected validators reconstructing the operation list, running the same transactions in the same order and producing the same collation root. Validators sign their produced root and commit to it by approving the collation. After n-number of validators have all committed the original collator commits the coalition and if all items match then the collation root is added to a dual layer accumulator and all variable updates are now valid on the network.

My currently running theory is that if n-number of randomly selected validators all agree then this suffices for ‘proof of publication.’

Any unfinished collations can still be finalized in ‘Uncle’ status. These uncled collations participants can still get reward if they share their log accumulations with the collators of the next set.

If the items do not all match the validators are all at risk of being slashed. A truebit style arbitration can keep slashes from occurring.

Transactions are executed by collators using a modified VM. The structure of transaction contains:

- Signature of caller
- Caller address
- Contract Address
- Code Hash - IPFS hash where code can be retrieved from
- Contract Proof - proof that the code hash was previously deployed at that contract address.
- Nonce
- Timeout - Block that user wants the transaction to timeout.

Executors follow the following Procedure:

1. Validate the contract proof
2. Download contract code
3. Add code to VM
4. Execute function
5. SLOADs will require downloading proofs
6. Collators and Validators are responsible for ensuring they don’t use a proof for an ADD log that has been DELed in a more recent block. They must do this by collection cryptographically actionable promised from data shard providers or by storing the latest state themselves.
7. Collators and Validators are responsible for ensuring they don’t use a proof for an ADD log that has been DELed in a more recent block. They must do this by collection cryptographically actionable promised from data shard providers or

by storing the latest state themselves.

8. SSTORs will output:
9. A DEL log if the item previously existed
10. A ADD log with the latest nonce and block
11. A DEL log if the item previously existed
12. A ADD log with the latest nonce and block
13. EXTCODECOPY, DELEGATECALL, CALL, and CALLCODE will require downloading new code.
14. Main net code from chain
15. [XXXXXXX] Code from IPFS(where does proof come from?)
16. Main net code from chain
17. [XXXXXXX] Code from IPFS(where does proof come from?)
18. CREATE will create a new contract in the [XXXXXXX]
19. New contracts should add ‘galaxy’ in position 1 to the keccak used to generate the contract address such that there are no collisions with the main ethereum network.
20. New contracts should add ‘galaxy’ in position 1 to the keccak used to generate the contract address such that there are no collisions with the main ethereum network.
21. SELFDESTRUCT will DEL the contract creation proof
22. SLOADs will require downloading proofs
23. Collators and Validators are responsible for ensuring they don’t use a proof for an ADD log that has been DELed in a more recent block. They must do this by collection cryptographically actionable promised from data shard providers or by storing the latest state themselves.
24. Collators and Validators are responsible for ensuring they don’t use a proof for an ADD log that has been DELed in a more recent block. They must do this by collection cryptographically actionable promised from data shard providers or by storing the latest state themselves.
25. SSTORs will output:
26. A DEL log if the item previously existed
27. A ADD log with the latest nonce and block
28. A DEL log if the item previously existed
29. A ADD log with the latest nonce and block
30. EXTCODECOPY, DELEGATECALL, CALL, and CALLCODE will require downloading new code.
31. Main net code from chain
32. [XXXXXXX] Code from IPFS(where does proof come from?)
33. Main net code from chain
34. [XXXXXXX] Code from IPFS(where does proof come from?)
35. CREATE will create a new contract in the [XXXXXXX]
36. New contracts should add ‘galaxy’ in position 1 to the keccak used to generate the contract address such that there are no collisions with the main ethereum network.
37. New contracts should add ‘galaxy’ in position 1 to the keccak used to generate the contract address such that there are no collisions with the main ethereum network.
38. SELFDESTRUCT will DEL the contract creation proof
39. Tack on gas charges in local token

40. Tack on payments for remote data requests

[XXXXXXX] assumes a market based solution to distributing the storage of witnesses and uncled collations. One possible solution is proposed below:

Cryptographic Handshake for Witnesses

I need X -> Slash me if I collate and can’t provide a proof of paying for X

<- I have X : Slash me if I can’t produce a proof of X

Give Me X-> Slash me if I collate and can’t provide a proof of paying YOU for X

Provider can now withhold for a DoS attack but has little financial reason to.

<-Here is X : Slash me if someone else produces a different X with a better proof

Cryptographic Handshake for Preprocessed Collations

I need Collations ->: Receivers should be able to prove signature matches qualified collator

<- I have C(Op Hash) : Slash me if I can’t produce a collation

Give Me C(Op Hash)-> Slash me if I collate and can’t provide a proof of paying YOU for C(Op Hash)

Provider can now withhold for a DoS attack but has little financial reason to.

<-Here is C(Op Hash) : Slash me if the collation isn’t valid