ZKU Assignment 6

Name: Shyam Patel
Discord: L_guy#4297
Email: shyampkira@gmail.com
GitHub Repo: [Link]

Q1 Interoperability trilemma

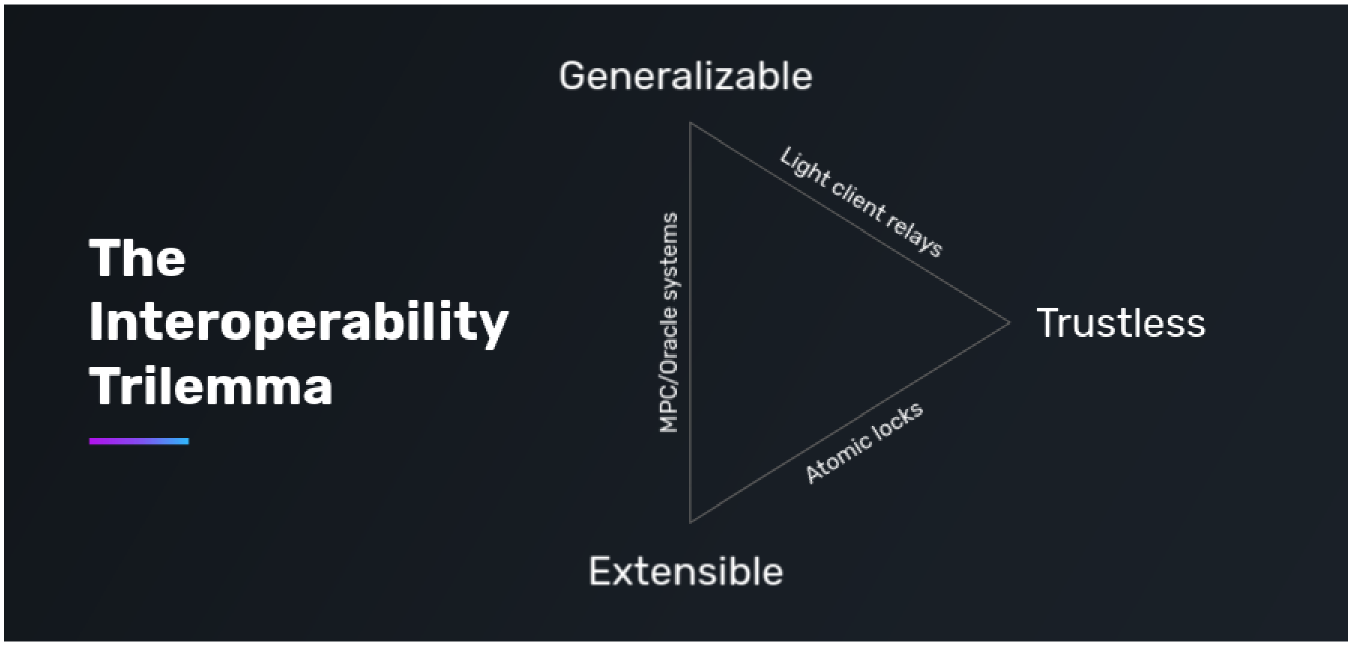
We talked before about the scalability trilemma where L1 must sacrifice either decentralization, security or scalability. In a cross chain world there is also a famous interoperability trilemma.

Interoperability protocols can only have two of the following three properties:

- Trustlessness
- Extensibility
- Generalizability

1. Explain each aspect of the interoperability trilemma. Provide an example of a bridge protocol explaining which trade-offs on the trilemma the bridge makes.

The Interoperability Trilemma:



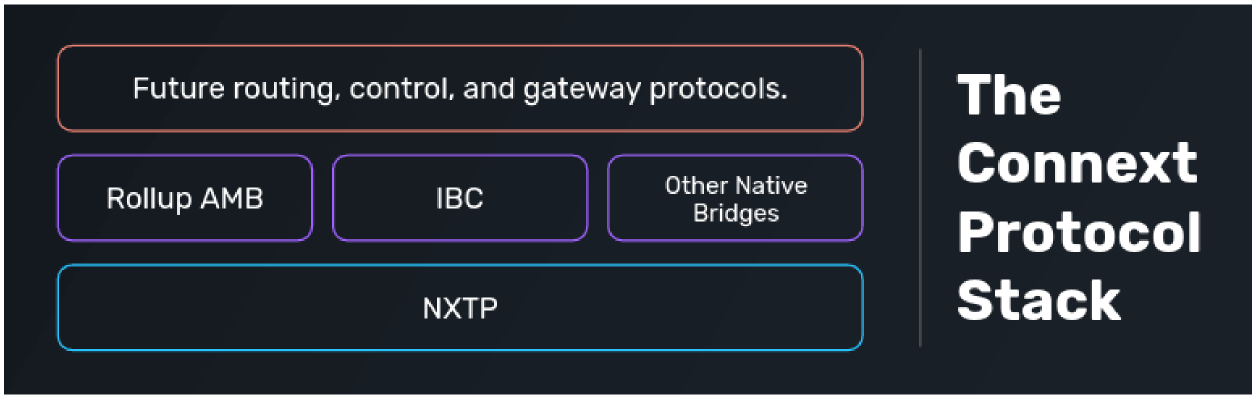
Similar to the Scalability Trilemma described in Vitalik's blog, there exists an Interoperability Trilemma in the Ethereum ecosystem. Inter-op protocols can only have two of the following three properties:

- Trustlessness: having equivalent security to the underlying domains. Maximizing the security of the system by disincentivizing validator collusion and making corruption attempts economically unfeasible.
- Extensibility: able to be supported on any domain or Seamless extension of the underlying protocol to other domains.
- Generalizability: The protocol's capability of handling arbitrary cross-domain data.

Light clients & Relays bridges (for example: Cosmos' IBC, Near's Rainbow bridge) are strong with Generalizability because header relay systems could pass around any kind of data.
They are also strong with Trustlessness because they do not require additional trust assumptions, although there is a liveness assumption because a relayer is still required to transmit the information and do not need any capital lockup. These strengths come at the cost of Extensibility.
For each chain pair, developers must deploy a new light client smart contract on both the source and destination chain, which is somewhere between O(LogN) and O(N) complexity depending on the underline chains.
This protocol also introduces latency and speed drawbacks caused by synchronizing data and validating proofs.

2. [Bonus] Are there any projects that focus on solving trilemma similarly like Ethereum solves the scalability issue? If yes, describe how it solves the problem.

Connext is one of the projects trying to solve this trilemma.
Like Ethereum that adds scalability via L2/sharding as a layer on top of an existing secure and decentralized backbone, Connext establishes the two most important properties first that are essential to the longevity of the protocol. In the case of interoperability, the two most important properties are maximum **trustlessness** and **extensibility**. The Connext NXTP offers these two properties and is specifically designed to be usable on any chain while still being as secure as the underlying domains.
In the next step, **generalizability** is added by plugging in natively verified protocols on top of NXTP. This adds a "Layer 2" to Connext's interoperability network.
By establishing maximum trustlessness and extensibility through NXTP and adding generalizability on top of NXTP, Connext can offer users and developers a consistent interface across any domain. At the same time, it empowers its users to "upgrade" their connection whenever the functionality of generalizability is available.


Future routing, control, and gateway protocols.
Rollup AMB
IBC
Other Native Bridges
NXTP
The Connext Protocol Stack

The Connext NXTP acts as the base protocol for Connext's interoperability network. It ensures maximum trustlessness as well as extensibility and allows natively verified protocols to be built on top of it. For this reason, the full Connext network will consist of an entire stack of protocols which includes NXTP but also generalized cross-chain bridges (specific to a pair of bridges), and protocols that connect the entire stack into one seamless system.

Q.2 Ultra light clients

This week we focused on how to achieve ZK interoperability between chains. How we can use Zero Knowledge Proofs in order to sync a light client faster. This is specially useful for mobile first blockchains like Celo, but is very useful for any blockchain.

1. Describe a specification for a light client based on zero-knowledge proofs. You should explain at least how to get the client synced up to the current state of the blockchain. Preferably go as far as explain how transaction inclusion proofs are generated too.

BLS:

A BLS digital signature— also known as Boneh–Lynn–Shacham - is a cryptographic signature scheme which allows a user to verify that a signer is authentic.
An important property of the construction (of BLS signature aggregation) is that the scheme is secure against a rogue public-key attack without requiring users to prove knowledge of their secret keys.

An ideal Zero-knowledge based light client will be one which requires downloading only the latest proof and verifying it to sync to the latest state of the blockchain.
Syncing to the latest state of blockchain for PoS chains means getting the trustable epoch committee (elected validator public keys), such that user's block inclusion proofs can be validated.

A ZK-based light client would include:

- Prover (validator/full nodes): generates a proof of committee transition from epoch i to j
    - Inputs: Epoch numbers i & j, committee (BLS public keys) of epoch i
    - Perform for each epoch block
        - Aggregate public keys from header according to bitmap
        - Check they are more than ⅔ threshold
        - Check the epoch number is epoch number + 1
        - Encode the epoch into bits and hash to group element
        - Compute and check BLS signature
    - Proof for epoch i+1 = Proof for epoch i + above operations
    - Aggregate proof either iteratively or recursively
    - Output: Proof that includes the transition from epoch i committee to epoch j committee

2. What is the relevance of light clients for bridge applications? How does it affect relayers?

Light clients allow a user to get sufficient evidence of a transaction's inclusion in a blockchain without having to operate a full node themselves.
Bitcoin does this with Bitcoin SPV.

Through this process, provers (full nodes) can provide verifiers (light clients) sufficient evidence to prove that a particular chain tip represents the longest (heaviest/most difficult) chain, without the light client putting trust in any 3rd party. Through a Merkle path to a transaction root in one of the block headers, a user can know that a transaction is included in a block in the longest chain.

3. Suppose code from Plumo is updated and is working in production on Celo. What would be the main difficulty in porting Plumo over to Harmony?

Celo and Harmony have a lot of common in underline technologies, specifically,

- POS based mechanism to select validators.
- Byzantine Fault Tolerance based consensus algorithm (Celo uses PBFT and Harmony uses FBFT).
- Leverage BLS multi-signature and need 2/3 validators' signature to validate a block.
- The validators won't change during a 24-hour's epoch interval. This also implies that the ultra-light client can efficiently pack all blocks in one epoch.

From migrating Plumo to work on Harmony perspective, the main difficulty is to change Plumo's circuit to fit Harmony's block structure, namely:

- Support sharding and 250 validators for each shard.
- Algorithm used by Harmony to create BLS multisignature.
- Implementation of recursive zkSnark to make sure the proof size is acceptable.

Q3. Horizon Bridge

Horizon is Harmony's bridge which allows crossing assets from Harmony to Ethereum/Binance and vice versa.

1. Check out Horizon repository. Briefly explain how the bridge process works (mention all necessary steps).

As described in documentation of horizon [README]

Bridge Components:

Used in Ethereum to Harmony flow:

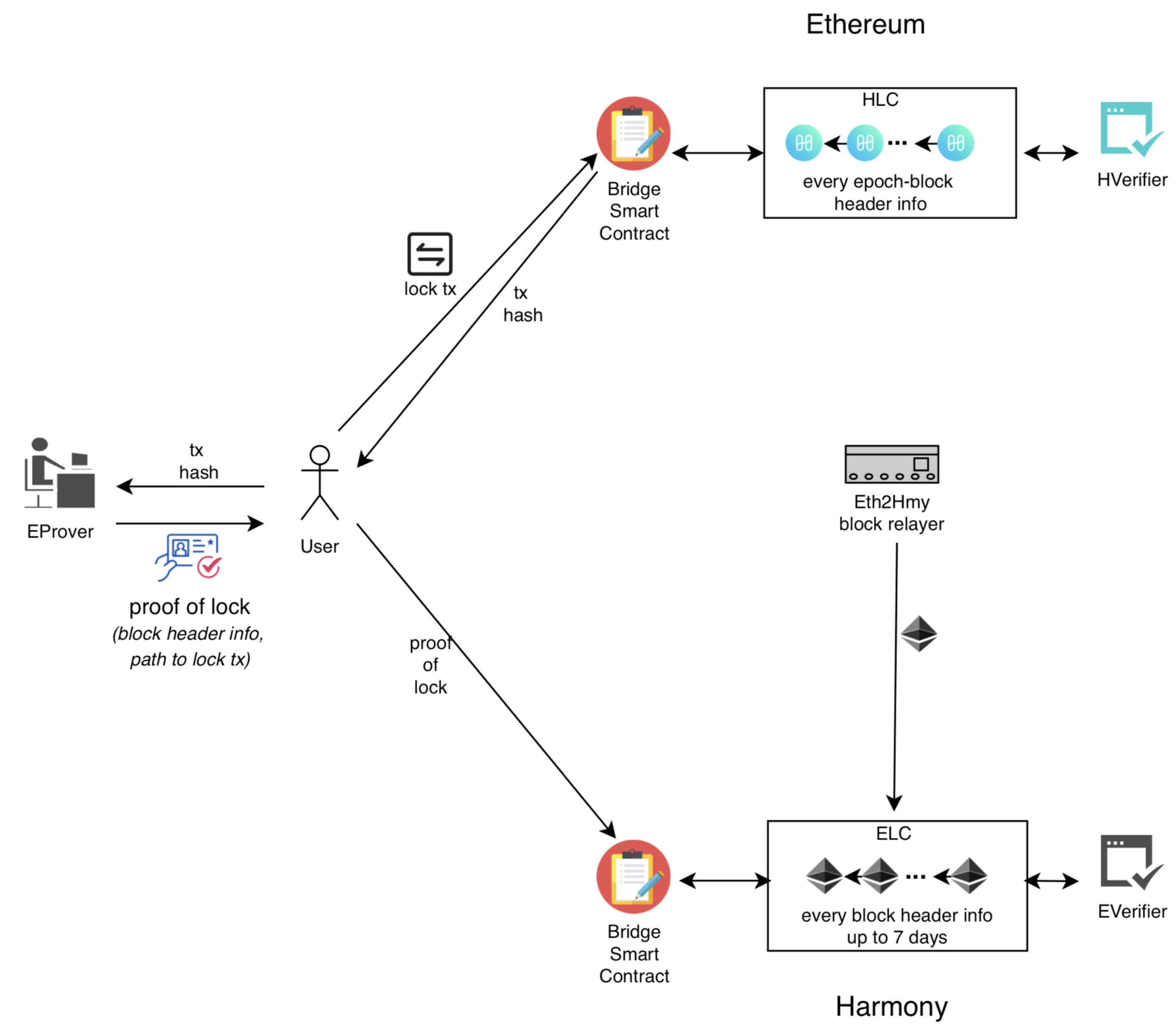
- Bridge smart contract on Harmony
- Ethereum Light Client (ELC) smart contract on Harmony
- Ethereum Verifier (EVerifier) smart contract on Harmony
- Ethereum Prover (EProver) is an Ethereum full node or a client that has access to a full node
- Ethereum Relayer relays every Ethereum header information to ELC

Used in Harmony to Ethereum flow

- Bridge smart contract on Ethereum
- Harmony Light Client (HLC) smart contract on Ethereum
- Harmony Verifier (HVerifier) smart contract on Ethereum
- Harmony Prover (HProver) is a Harmony full node or a client that has access to a full node
- Harmony Relayer relays every checkpoint block header information to HLC

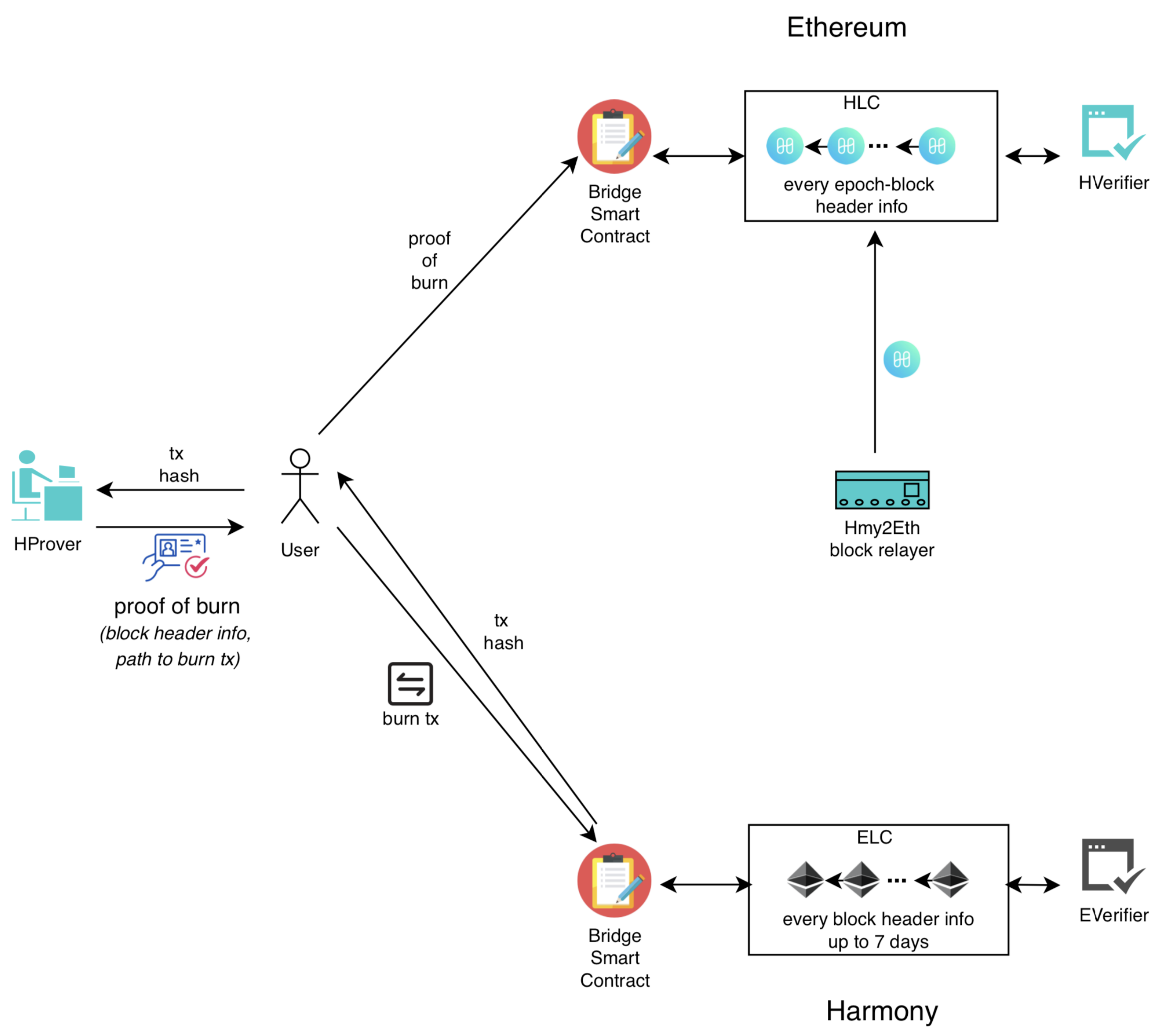
Ethereum to Harmony asset transfer

1. User locks ERC20 on Ethereum by transferring to bridge smart contract and obtains the hash of this transaction from blockchain
2. User sends the hash to EProver and receives proof-of-lock
3. User sends the proof-of-lock to bridge smart contract on Harmony
4. Bridge smart contract on Harmony invokes ELC and EVerifier to verify the proof-of-lock and mints HRC20 (equivalent amount)



Harmony to Ethereum asset redeem

1. User burns HRC20 on Harmony using Bridge smart contract and obtains the hash of this transaction from blockchain
2. User sends the hash to HProver and receives proof-of-burn
3. User sends the proof-of-burn to bridge smart contract on Ethereum
4. Bridge smart contract on Ethereum invokes HLC and HVerifier to verify the proof-of-burn and unlocks ERC20 (equivalent amount)



a. Commented the code for:

- harmony light client
- ethereum light client
- token locker on harmony
- token locker on ethereum
- test contract

b. Why HarmonyLightClient has bytes32 mmrRoot field and EthereumLightClient does not? (You will need to think of blockchain architecture to answer this)

Ethereum is POW based, adding mmrRoot in EthereumLightClient to support transaction inclusion check, the best effort light client can do is to implement a FlyClient like mechanism. This requires replayer to send Log(N) (way better than SPV) of block headers to the smart contract and do verifications (each block will need to do a mmr root inclusion verification and block POW verification) inside the smart contract. Such verification (> 700 blocks) inside a smart contract is prohibitively expensive and could break the maximum transaction gas limit.

Harmony is POS based and a block is verified by checking BLS multi-signature. A good feature of BLS multi-signature is it can combine multiple messages and each message with a different set of singers in one signature verification process. Leveraging such capability, the light client only needs to verify 1 signature every epoch (an epoch contains 32768 blocks or ~18 hours) and only needs the relayer to send 1 block per epoch. This makes the verification of signatures inside a smart contract feasible. This also enables mmrRoot (for an epoch or a checkpoint) to be deployed to support faster transaction inclusion checks. In Horizon, a maximum of 16384 blocks can be handled in one update.

2. Horizon still doesn't use zk-proofs in order to speed up light clients. What changes would you need to make to the code in order to apply initial state sync through zk-snarks? Provide pseudo code of improved version of light client.

Similar to Plumo, we can generate a recursive zk-snarks proof from genesis block to the latest epoch, for each epoch the circuit (C1) need to prove:

- Aggregate public keys from header according to bitmap
- More than 2/3 validators signed, and the signature is valid
- Epoch number = previous epoch number + 1

Then to generate recursive zk-snarks, we need circuit (C2) to recursively compute the proof (from first epoch to latest epoch) to prove:

- Previous epoch proof is valid
- Current epoch is valid The input parameter would be mmr root of genesis block, and output would be the new mmr root of last block in the latest epoch.

Then for the rest block (block generated after the latest epoch), the relayer calls a prover to make a checkpoint proof using a circuit similar to C1. Then periodically, the relayer will call the prover to make checkpoint proof and update mmr root. Major functions in `HarmonyLightClient.sol` that would need changes:

```
function initialize(bytes32 memory firstMMRRoot, Proof memory zkRecursiveProof)
   external
   initializer
{
   MMRRoot = firstMMRRoot;
   require(verify(zkRecursiveProof, firstMMRRoot), "invalid recursive proof");
   MMRRoot = zkrProof.output["mmrRoot"];
}

function submitCheckpoint(Proof memory zkCheckPointProof)
   external
   onlyRelayers
   whenNotPaused
{
   require(verify(zkCheckPointProof, MMRRoot), "invalid recursive proof");
   MMRRoot = zkCheckPointProof.output["mmrRoot"];
}
```

Q4. Rainbow Bridge

1. Comment code implemented in NearBridge.sol. (Note: that they're using fraud proofs and not zk proofs.)

Link to commented code: [NearBridge.sol]

2. Explain the differences between Rainbow bridge and Horizon bridge. Which approach would you take when building your own bridge (describe technology stack you would use)?

- Rainbow Bridge is generic. Horizon Bridge is ERC20 specific. This induces a higher latency for the Rainbow bridge.
- EthOnNearClient contract then memorizes the merkle roots of the DAG files for the next 4 years upon initialization. On harmony, there is a far shorter cycle (in days).
- The current Rainbow Bridge does not implement incentives for the maintainers who pay for the gas by relaying the headers. This is not an issue on Horizon, as horizon has no need for constant updating.

RainBow and Horizon have more in common than differences. They both have key components as: smart contract, relayer and prover. The extensibility of RainBow allows more specific provers to be built that could satisfy more use cases, given it has a better trustless design. RainBow has a kind of better implementation as per my knowledge but I would've to explore more to know for sure. A hybrid approach by leveraging the advantages of RainBow, Horizon and Plumo would be more suitable in this case. Borrow the architecture from RainBow/Horizon and implement recursive Plumo like zkSnark for block/epoch verification. Or something else completely different and better.

Q5. Thinking In ZK

1. If you have a chance to meet with the people who built the above protocols what questions would you ask them?