# Gaming RollUp

## Requirements

### 1.1 Transaction types

From blockchain perspective, Gaming could be half-decentralized. Because each game has an operator, who’s a major participant in the ecosystem. Game operator defines the rules in game and on chain, also manages the game. We can define 2 kinds of transactions for gaming:

* **Game transactions**
* Related to some specific game, it should follow the rules in game and smart contract. Like a player get some weapons , or some trading inside the game.
* **This kind of transactions should be endorsed by the game operator**, because they need to be approved by the operator to take effect in the game. The way to endorse for the transactions is to sign on it. This kind of transaction needs multi signature - by player and game operator. Game operator could sign on many transactions at one time.
* I**f we don’t need endorsement for transactions, just treat them as global transactions, it’s easier.**
* **Global transactions**
* Not related to specific game, like DeFi, GameFi, trading between different games.
* This kind of transactions don’t need be endorsed by game operators, it’s just like normal transactions on Ethereum.

### 1.2 Transaction requirements

Some transactions like trading in the game, players are waiting in front of the screen, so it should be completed within 1-3 seconds. Some other transactions like recording the result of a fight, players don’t care about when it’s completed, they may even don’t noticed it. This kind of transactions can be bundled. So generally, **we should confirm a transaction in 1-3 seconds**.

Some transaction may contain attachments as part of it, like image of the skin, proof of a medal, etc.

### 1.3 Throughput requirements

We maybe supporting thousands of games , with millions of players at the same time. So how to build a scalable chain is important.

## Architect

### 2.1 Overview

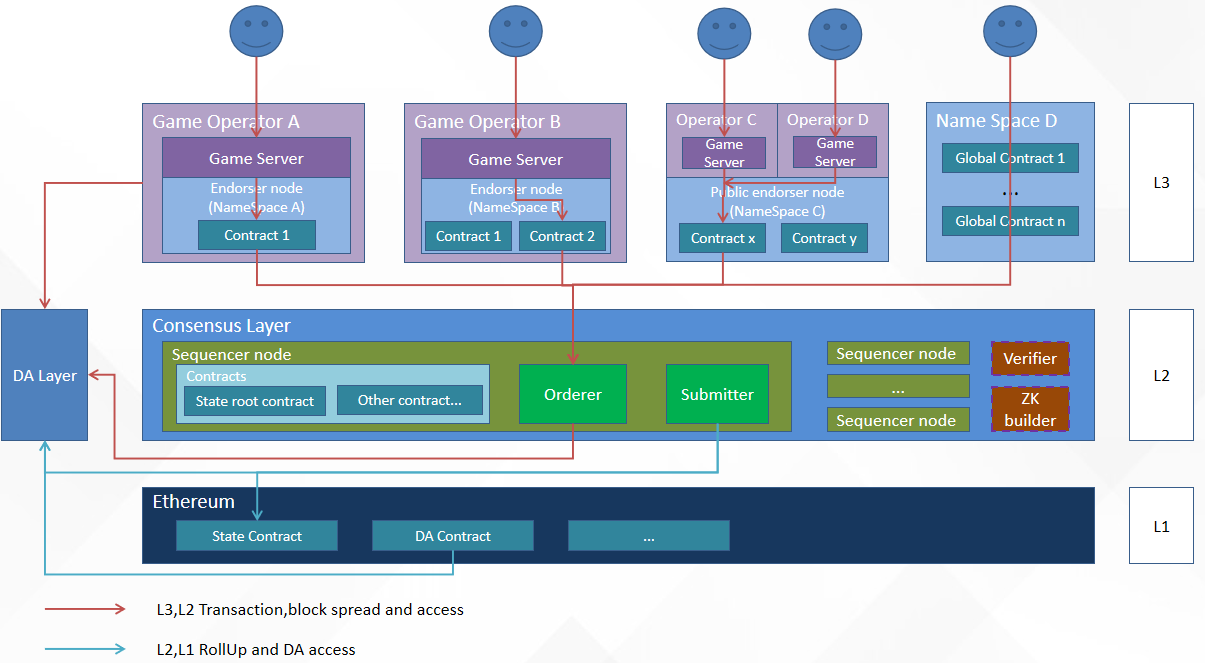


Fig 1. Architect of Gaming RollUp

The architect refers to the design of Fabric,Celestia, and Eigen DA. As in fig 1,we have 3 layers, we can separate it into 2 parts:

* L3->L2: it works like Celestia, the difference is, L3 will submit the state root to L2, because we need to get consensus to the L3 states, not only the transactions.
* L2->L1 RollUp: we can use OP rollup or ZK rollup.

L1 is Ethereum, we’ll deploy some contracts for the rollup.

L2 is the consensus layer, we have many sequencer nodes here, it’s decentralized. Sequencer has 3 parts:

* EVM: it runs L2 contracts, they’re all system contracts, like gathering the states of L3 and handle the stakes, e.x. EigenLayer.
* Orderer: it collects the L3 txs , order them, build the block and get consensus to it.
* Submitter: it works as part of rollup. We also have Verifier and ZK builder, we’ll talk about them later.

We also have Verifier and ZK builder at L2.

L3 is the execute layer. It’s sharding by namespace, like Celestia. In a namespace, we can have at least one node , it executes the tx and stores the state. If we have more than one node, the state on them must be same. We could deploy many contracts in one namespace, and each contract can have its own endorsement policy. The namespaces are independent to each other because they’re in different shard, they have different state root. We have many kinds of shards in L3, the shard operators can decide it. Here please pay attention to the namespace D , all contracts here are global contracts, so they don’t need endorsement, it’s just like the Ethereum case.

We also have the DA Layer, the block data, tx data, roll up data are all stored and be served on it.

### 2.2 Main process

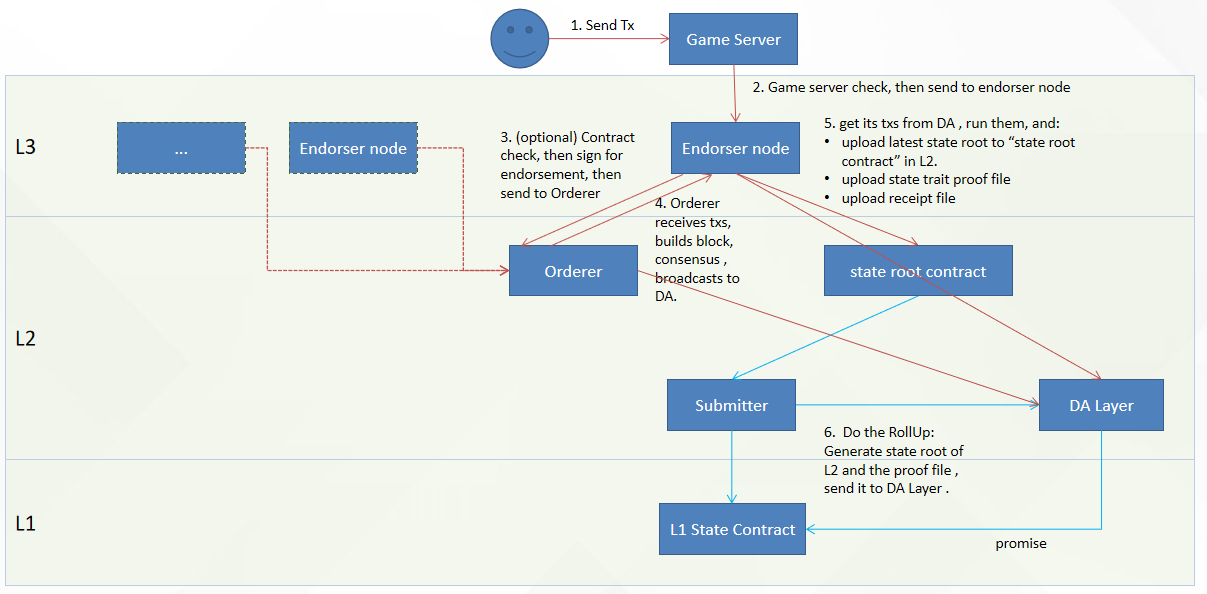


Fig 2. Main process of a transaction

Here’s how a transaction is processed for a game transaction. It has 2 main stages. Stage 1 is L3 and L2 processing,with red arrow, stage2 is RollUp to L1, with blue arrow.

**Stage 1 (L3->L2):**

1. Player sends an transaction in the game.
2. Game server receives and does some checks on it, if looks good, it will be send to endorser node.
3. Endorser node will pre-execute the transaction to check whether it’s legal, if it looks good, it will endorse for it, that means, sign it with endorser’s private key.
4. (To improve throughput, and could give suggestions of the order of the txs, endorser node is encouraged to endorse as many transactions as possible in a single message during one block time, and give a suggested order. Because L3 endorser nodes could only update contract state after a new block is confirmed by L2 consensus layer. So handling many transactions in one message has the same effect with handling each transaction in each message.)
5. In L2 Consensus layer, Orderer will receive endorsed transactions from all L3 endorser nodes and global transactions from user. Orderer will put all transactions together and decide their order, and get consensus. Then broadcast the block to DA Layer.
6. The block format should be well designed that DA layer can easily split it , because L3 nodes normally only interested in the transactions of its namespace, so DA node needs to split the block and return the required data for L3 node requests.
7. After get ordered transactions from DA node, endorser node will execute them, update the contract state, and :
   1. Send a transaction to “state root contract” to update the latest L3 contract state root, it includes the cid of (2)(3).
   2. Receipts of the transactions. Because some transactions maybe invalid , and the events and tx execute results needs be recorded and can be queried (DApp front end needs it). We can use DA layer to supply it.
   3. Generate state transit proof file, upload to DA Layer. This contains the pre-state of the contract, executed valid transactions in current block, and post-state of the contract. It works like fraud proof.

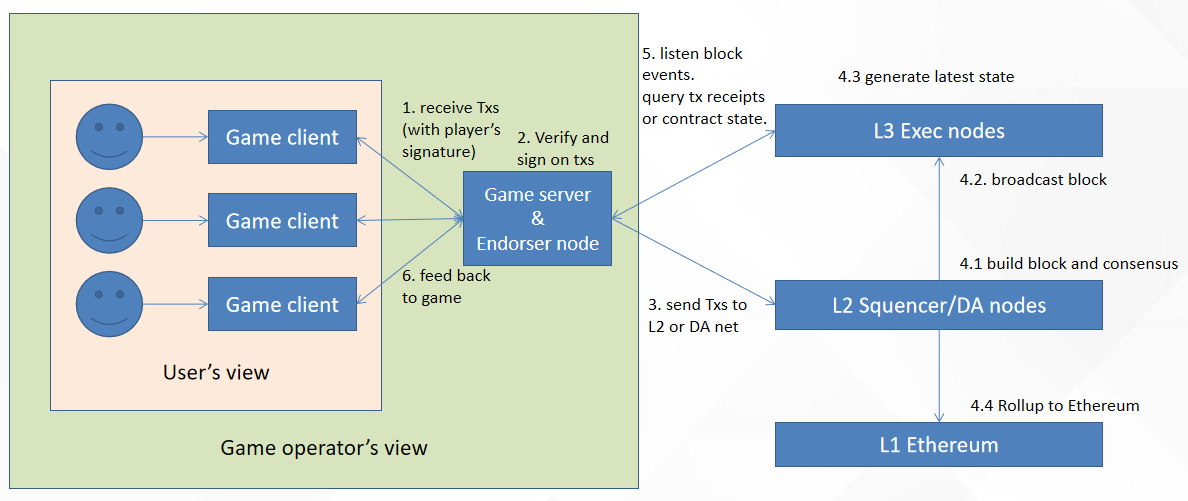
**Stage 2 (L2->L1):**

1. L2 Submitter will do a RollUp to the Ethereum L1. We can use OP RollUp or ZK RollUp (ZK is preferred if available) .The RollUp has below contents:
   1. The proof file (via DA Layer), it’s actually made up by the links of :
      1. The block generated in 4.
      2. All the receipt files supplied by endorser nodes in 5(2)
      3. All state transit proof files supplied by endorser nodes in 5(3)
   2. Latest state root.
   3. Other proofs and data needed by RollUp or eigenDA .

For global transactions, it’s easier, transactions are directly sent to L2 Orderer, we just omit the endorsement part.

In ideal process, the above processes should align with L2 block. Whenever a new block is built on L2, all following works be done to reflect the state changes it brings to the system. But unfortunately, all parts work asynchronously, some L3 nodes may even off line. So actually we do is: L3->L2 and L2->L1 work separately, they don’t care each other’s situation. So in the L3->L2 state transit proof file in 5(3), and the L2 state root contract, we need to record which block this is for. Also in the L2->L1 proof file in 6(1), we need to record the block number of each file in DA and state root.

Besides, below pic shows the view from game operator’s view, it’s the same as above, just game server and endorser node are put together, which maybe more possible in implementation.



### 2.3 Block structure

### 

Above is the block structure. It has head and body.

#### 2.3.1 Head

Comparing to Ethereum block, the head added:

* SequencersCommit.
* This’s a Merkle root of sequencers’ id. Notice that here we don’t use KZG commit, because KZG commit can only prove a node is a sequencer, it can’t prove a list is the “full” list.
* TxMode. 1: full tx body, 2: tx hash,...
* In a block, at the beginning we may have few txs, but as we have more and more txs, the block body will bloat, the bandwidth will be the bottleneck of consensus. So this field defines how to explain the tx content in the block. If 1, it’s the full content of tx, if 2, it’s just a hash of the tx. Later we may have other solutions for store tx in the block.
* TxRoot
* It’s the root of the txs, we can use KZG commit here. Notice that the body is organized by name spaces, each name space has its tx merkle root. The TxRoot in head is the root of the “roots in name space”.
* Notice that no matter TxMode is 1 or 2, the TxRoot will be the same.

Notice that the head don’t have state root, receipt root, gas related fields, because we can’t get them when the block is built.

#### 2.3.2 Body

Body is organized by name spaces,name spaces are ordered by it’s id.

For each name space, it has below contents:

* Name space id: fixed length digits.
* Length: Length of current name space contents.
* Tx root: root of the txs, KZG commit maybe used.
* DA commit: If TxMode is 2, we only store tx hash in the block, so we need to find a way to make sure L3 nodes could get the full tx from DA node. So we may need the DA nodes to give a commit that it will supply the tx body (maybe sequencers will only package the committed txs into the block) .
* Tx list: list of the txs.

### 2.4 Account,Contract and endorsement policy

For an account, on the different sharding, its nonce is different. But that will not cause any problem, it’s the way RollUp works. It could be seen as different accounts with same private key on different shardings.

Contract’s endorsement policy should be registered on L2, and can be updated. Global contracts’ endorsement policy can be none. Endorsement policy can be combination of some accounts. Like :

* All of [...]
* Any of [...]
* (A AND B) OR C

The endorsement policy part referenced Fabric :

<https://hyperledger-fabric.readthedocs.io/en/latest/policies/policies.html#chaincode-endorsement-policies>

### 2.5 Contract interactions

This part we’re different from Celestia (https://arxiv.org/pdf/1905.09274.pdf 5.1.1 Cross-Application Calls). Because Celestia is sharding by dapp, but we’re sharding by namespace. so we have different behaviors, but from deeper, we’re the same (we can think on Celestia, it’s sharding by dapp).

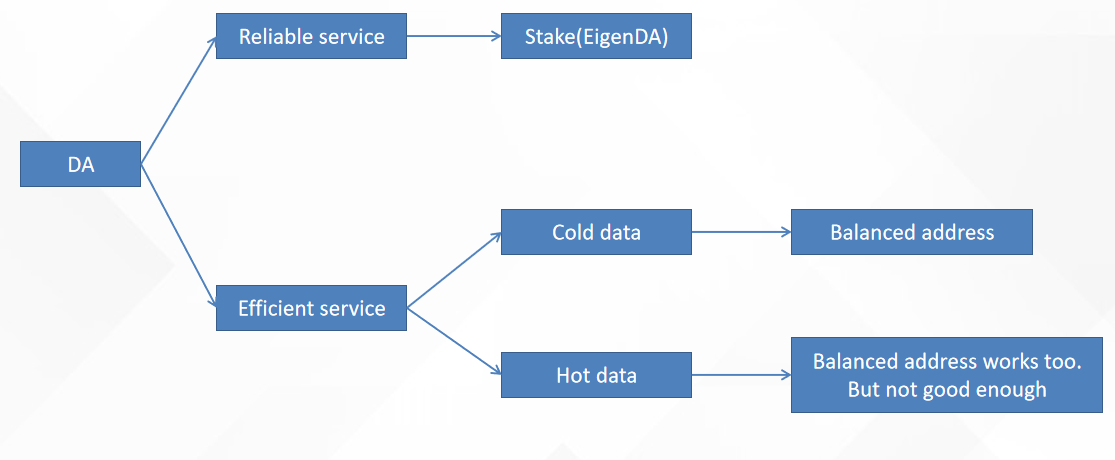
We should distinguish the transaction call and contract call.

* + The transaction call will be packed into block, be ordered with other transaction calls, and all L3 nodes can see it, that means all namespace of L3 can see it.
  + Contract call is : When a contract is executing , it may call other contracts, in this case, these 2 contracts must share the same state root, means they must be in the same namespace.

So , if a contract is going to interact with another contract, if they’re in the same namespace, it’s contract call, just like on Ethereum, it can directly do it. If it’s on different namespaces, the caller must send a new transaction call to do it, the new transaction call should be packed into the subsequent blocks and be handled by the target namespace, it’s actually an cross chain call.

## DA Layer

### 3.1 requirements



We need to supply reliable and efficient services.

Reliable service can be supplied via stake (EigenDA).

On efficient service , we should consider cold data and hot data separately.

* Cold data is storage sensitive, it’s not urgent, user can wait some time to get it. it’s a m of n situation (m can be 1)
* Hot data is bandwidth sensitive, users wish to get it asap, so we may wish all DA nodes hold it. (when hot data become cold, DA node can remove it).

We may need different solutions for cold data and hot data. Balanced address should be a good solution for cold data. It could also improve the performance of hot data, as the L3 nodes are name space sharding.

### 3.2 Balanced address solution

#### 3.2.1 Target

We use IPFS to build the DA layer.

IPFS uses DFT, In DFT, the basic idea is the data should be stored by the node whose node id is close to the data CID, this is naturally a sharding solution for data. The storage and bandwidth are sharded. We could encourage some IPFS nodes to store and serve some data, and other nodes to serve other data by economic awards.

### 3.2.2 node addresses evenly distributed

The 1st thing is to make the IPFS node id located even in the address space. The IPFS node address space is 2^256 , it looks like :



We wish the IPFS nodes can be located evenly in the address space. We define “generation” as how many digits counting from above. Generation increases when node number grows. Ex. When we have 2 nodes, we have 0 generation, because the highest digit can be “0” or “1”. when we have 4 nodes, we have 1 generations, because the highest digits can be “00”,”01”,”10”,”11”.

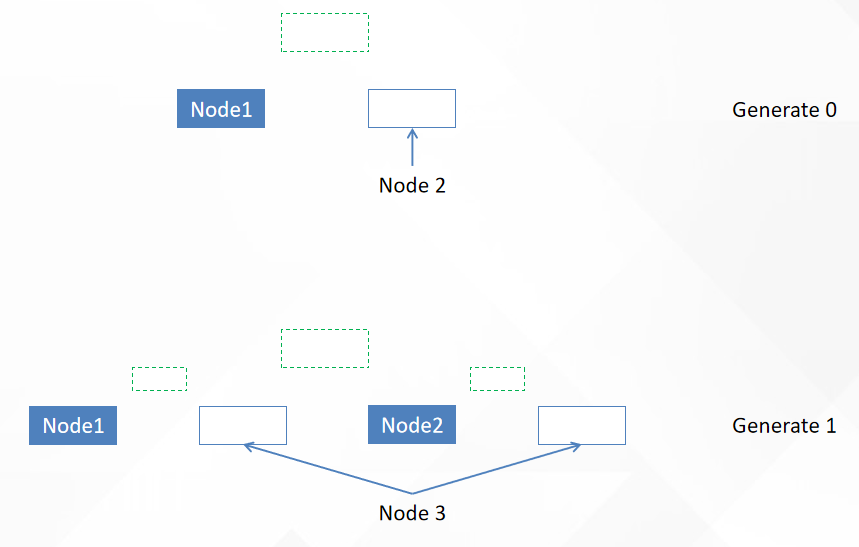


In every generation , the effective node address part is :





When a new IPFS node joins in to the p2p net, it should adjust its address, make it located on an opening position in current generation. By doing this, we can make sure the node addresses are always balanced.



#### 3.2.3 Award weight

Assume for a file slice, we wish nodes to store and supply service for it. So we need to make sure they can get most awards in mining if they finished their job.

Rule: The nodes whose addresses are most close to the slice CID, can get more awards.

Here we define “CommonPrefixLen” means how many highest bits of the node address is the same with the slice CID. And the the max commonPrefixLen is n (If generation is more than n, we don’t distinct them). The award weight of a node is .

For example, We have 16 nodes (generation =3, it begins from 0), we wish at least 4 nodes to save the slice. The slice CID begins with “01”. All 16 nodes save the slice ,how many award should each node get?

In this case, n=2, so we can have , 3 kinds of award weights.

The nodes whose addresses begins with “01” , has commonPrefixLen =2, its award weight is =4

The nodes begins with “00” has commonPrefixLen=1, its award weight is =2

The nodes begin with “1” has commonPrefixLen=0, its award weight is =1

Please check fig 3 for it.

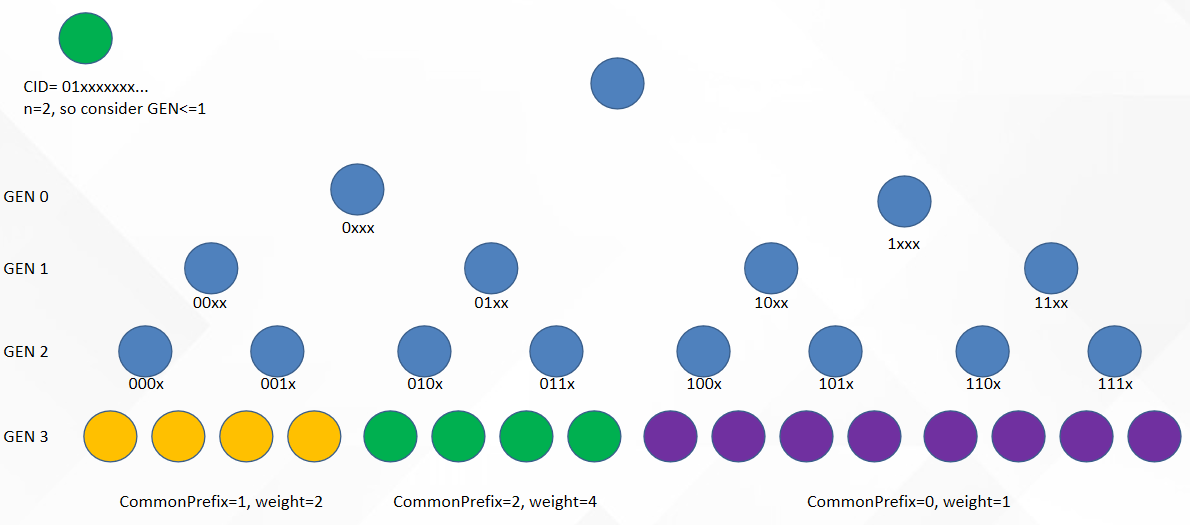


Fig 3 CommonPrefix and award weight

So we have 4 nodes weighted 4, 4 nodes weighted 2, 8 nodes weighted 1. the total award base should be 4\*4+4\*2+8\*1=32. Every green node gets 4/32 of the award, every yellow node gets 2/32 of the award, every purple node gets 1/32 of the award.

In this way, green nodes will try their best to save the slice. The purple nodes, if their storage and bandwidth is not that good, they will drop this slice, and try to store the slices located in their space. So we can get most efficient storage network.

The endorser nodes local connection to the DA Layer should be even aligned, so they can have as many generation connections as possible. When an endorser node try to get a block slice begins with “01”, it should directly ask from the green nodes, because they have the maximum probability have that slice.

### 3.3 DA service check

DA should supply reliable service. But there’re 2 possible risks:

* 1. Although EigenDA node committed to store and serve on datablob, it’s difficult to check it. Because when someone can’t get data from EigenDA node, it’s hard to prove whether it’s network issue or DA node just dropped the request.
  2. DA node should provide long time service on the datablob it committed, but how can we check and make sure it didn’t delete the data which’s very old?

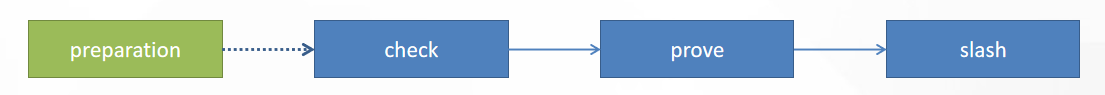
Here are some possible solutions:

* 1. DAS can solve the risks for blocks, but for other data, it may not work very well. And it’s too heavy.
  2. EigenDA should have its solution, but I don’t know it.
  3. I’ll give a solution, it can help on the 2nd risk.

#### 3.3.1 procedure

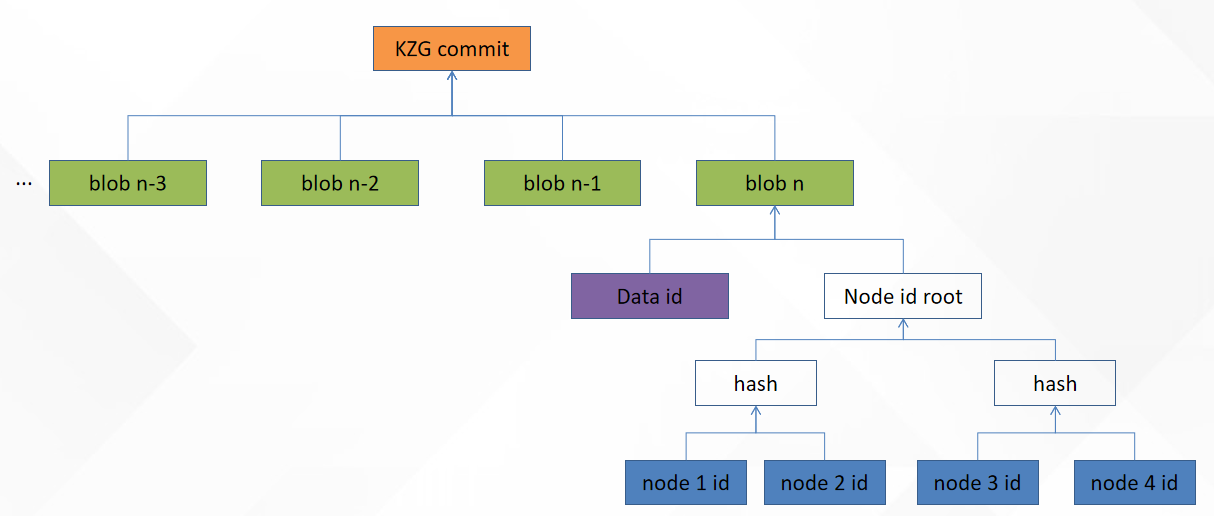
We can run a DAProof contract on L2 for this task. Assume the datablob served by DA node is indexed.

We have 4 steps to do the check.



**A) Preparation**

When DA node committed to serve on the data (see [proposal](https://github.com/windranger-io/public-docs/blob/main/research/R002.md)) , we could store a proof in the DAProof contract that can prove which nodes committed to store current datablob. it looks like below:



Current L2 is trying to rollup block n, the corresponding datablob is “blob n”. there’re 4 DA nodes committed to serve on “blob n”. We just need to update the orange “KZG commit” in DAProof contract.

Later when this blob is selected to be checked, sequencer could just upload the node list (the 4 blue id) and a SPV proof to the DAProof contract, because we already know the “Data id” (it’s selected to be checked), it’s very easy to prove the node list is the right list to be checked via the root.

Notice the orange proof is a KZG commit, because “n” should be very large , KZG proof for blob n should be very small and cheap to verify comparing to SPV proof. And the “Node id root” should be a merkle tree. Because normally the DA node id list shouldn’t be large, so merkle proof may works well here. It’s just a balance of cost.

1. **Check**

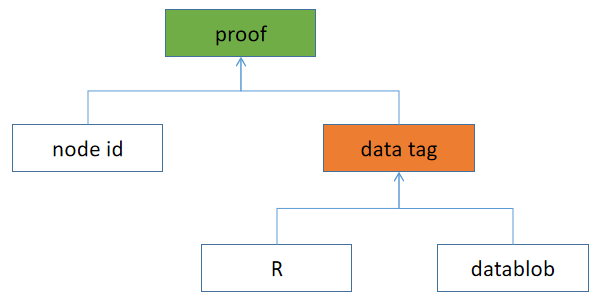
The check regularly happens (e.x. every 600 blocks).

Assume the latest datablob index is “N”. We generate a random number ,named “R”. We calculate “R mod N”, the remainder is the blob index which need be checked, let’s name it “blob i”. Sequencer should upload the node list who committed to serve “blob i” to DAProof contract, according to the “preparation step”, it’s easy to prove that’s the right list.

Extend: If we assume every blob has a weight, we can increase the weight of the “newer” blobs. E.x. the latest 1% blobs have a weight of 100 for each, latest 1%-10% blobs has a weight of 10 for each, all other blobs have a weight of 1. in this way, new blobs are more likely to be checked.

1. **Prove**

The proof looks like this:



R is the random number used in “Check” step, it must be on left of datablob (to make sure DA node stored the data itself, not just a hash). So the “data tag” should be the same for every DA node. Because every node’s id is different, so their proof is different. So no one could copy and use other node’s proof.

There’re 2 steps for the DA nodes to do:

1. Upload proof to the DAProof contract in limited time (e.x 40 block time).
2. After all proofs uploaded, any of the DA nodes could upload “data tag” to the DAProof contract. So the contract could verify each node’s proof. If one node don’t agree with another node’s “data tag”, he/she can upload his/her “data tag” to replace the previous one, we must make sure we use the same “data tag” to check all proofs. But if this happened, it means someone is lying.
3. **Slash**

If a DA node can’t upload proof, or it can’t be verified, it should be slashed.

#### 3.3.2 Risks and limitations

**Value**:

It costs much less resource than DAS

It could make sure the DA nodes store the blob in the long run. Because storage cost is much cheaper than be slashed , DA nodes don’t need to take risk.

**Limitations**:

This solution could only help to confirm the DA nodes stored the datablob. But if some DA node is malicious, it stores the data but don’t supply it to the requester, this solution can’t help.

Because storage is cheap, I’m not sure whether this is needed.

**Risks**:

If DA node is not decentralized enough, this solution may fail. Because in “prove” step , if all DA nodes are controlled by the same one, even they all deleted the datablob, they can collaborate to pass the check.

If in “prove” (2), different DA nodes supply different “data tag”, it means some one is lying. We should check out who’s lying.

* 1. An easy way is to ask all DA nodes upload the “data tag”, if most are the same, we believe that’s the right value. But if malicious collaborate, we maybe cheated.
  2. Another way is to use datablob id. The datablob id may contain the hash of the data(I guess), so we can directly verify it. But the blob maybe large, how could we do it in a decentralized way?

#### 3.3.3 Random number generation

In step “check”, we need a way to generate a random number easily, here’s a possible way.

**Preparation:**

* Find a signature algorithm which for the same message, signature is always the same. (e.x. RSA)
* Sequencers exchanged their signature public key.

**Procedure**:

1. Assume we have a random number in the last block. When a sequencer is building the next block, it will sign on the last random number by its signature private key. The signature will be the random number in the current block.
2. Other sequencers will check the signature by the signature public key they stored, and sign on it when they’re doing BFT, the random number confirmation will be part of the block consensus.

The 1st random number could be the signature on genesis block.

In this way, we can get:

* Each block will have a random number easily.
* This random number could be used for DApp, but can’t for system level.
* This random number could not be predicted and controlled in most cases on a decentralized chain. When a sequencer continues produces block, it could predict the random number, but it can’t control it.

## 4.Economic

### 4.1 Participants

We have many participants:

* Player/Game operator: Pay for the tx fee.
* L3 nodes: Execute the txs in the block, and upload the state root to L2 and proofs to DA.
* Sequencer: Build the block
* Verifier: Verify the state transit proof supplied by L3 nodes.
* Submitter: collect info and run the RollUp from L2 to L1.
* ZK Builder: built the ZK proof if ZK RollUp is used.
* DA node: serve for data availability.

**L3 nodes:**

It may pay for the tx, or the DA node because it uses their service.

I think it shouldn’t be payed in the platform. It’s a consumer from the chain platform view. It should get profits from its user via DApp, from game players.

**Sequencer**:

Node should stake to become sequencer, it supplies security and consensus to the platform, it’s the most import part in the platform.

It receives gas from the txs and can get block awards.

**Verifier and Submitter:**

They could be part of sequencer, so we don’t give extra awards to them.

For verifier, it should verify the state transit proof of the blocks it generated.

For submitter, we’ll consider who do the rollup in other chapters.

**ZK Builder**:

It’ll have a special market , please refer to the “[ZK RollUp suggestion](#_5.ZK RollUp suggestion)” chapter.

**DA node**:

It gets reward when committed to serve on some datablob.

### 4.2 Design

We have 2 Tokens, for gas and data service. Data service maybe handled by EigenDA, if it could work well, please ignore DT below.

For easy description, we call gas token “GT” (game token) and data service token “DT”(data token).

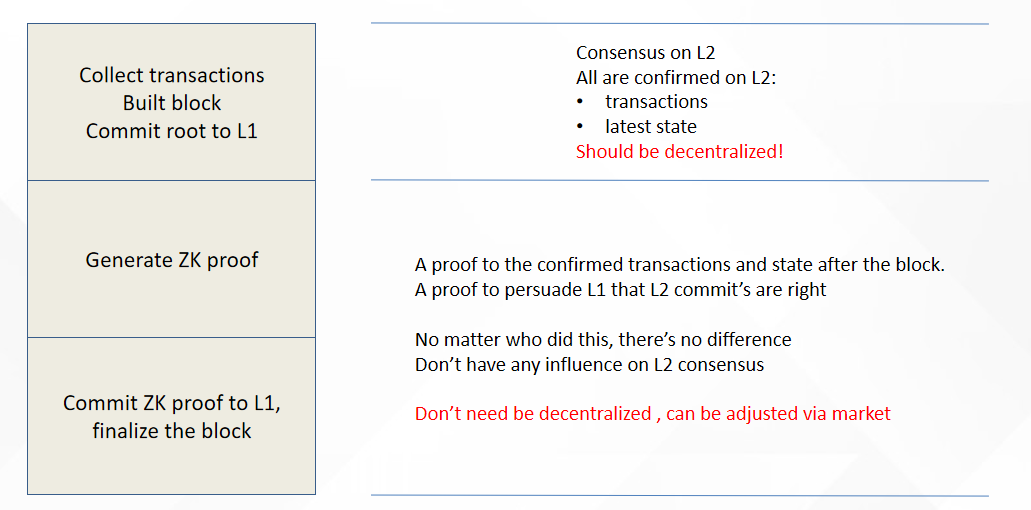
Both GT and DT are generated on L2 when building new blocks. They’re block awards. And both have pre-mint and airdrop ,and have token market to buy and sell.

GT could be used as gas, when player/game operator send tx to L2, they need to pay for the tx. The gas is calculated on data size of the tx. The sequencer who generates the new block could get the gas and block award.

DT is payed to DA node. When sequencer sends block data to DA node, or L3 nodes send data to DA node, they need to spend DA on it. We may have a public price on every byte and desired node service number. If the datablob to serve is larger, or the desired service node number is bigger, the cost should be more. When DA nodes committed to serve for the datablob, the nodes could get the service fee. We should consider how to balance the desired service node and actually commit node. And when a DA node quit, how to make sure data is served by other nodes.

## 5.ZK RollUp suggestion

### 5.1 overview



I think the main process of ZKRollUp could be divided into 3 stages. The 1st stage is the consensus on L2, it includes collecting the txs, building the block, generating new state, and committing the roots to the L1. This part is most import because the transactions and latest states are confirmed and fixed after this stage. The other 2 stages must follow the results of this stage. So this stage should be decentralized.

The 2nd stage is to generate the ZK proof for the results of stage1. the 3rd stage is to commit the ZK proof to L1, finalize the block. These 2 stages can also be seen as one. The target is to persuade L1 that the txs and latest state that committed in stage 1 can be trusted. No matter who build the proof and commit it, there’s no difference, and no influence on the result of 1st stage. So, I think this part don’t need be decentralized, we can run a market to auction the right to create and commit the proof.

### 5.2 Decentralization on stage 1

I have a specific doc about Decentralizing ZK Rollup, please check it [here](https://github.com/maodaishan/eos_doc/blob/master/Solutions%20and%20designs/Decentralizing%20ZKRollUp.docx).

It’s just the same as all other chains, currently I don’t have innovation on it.

For consensus, normally we can use POS , so stage and BFT is needed. Here we can consider EigenLayer.

Anti MEV, I think PBS can be used here. We’ll have many proposers and builders.

Anti Censorship, I think CRList can be used.

The process is as follow:

1. For each block, one proposer is selected (via stage weight).
2. The proposer/verifier give a CRList containing the txs he/she think should be included in the block.
3. Each builder can produce a candidate block, the block must contains the CRList. The Tx list is not publicly shown. The proposer leader selects one block without knowing the tx list.
4. All proposers vote to the block, if more than 2/3 signed, it’s used, if not, start a new round.

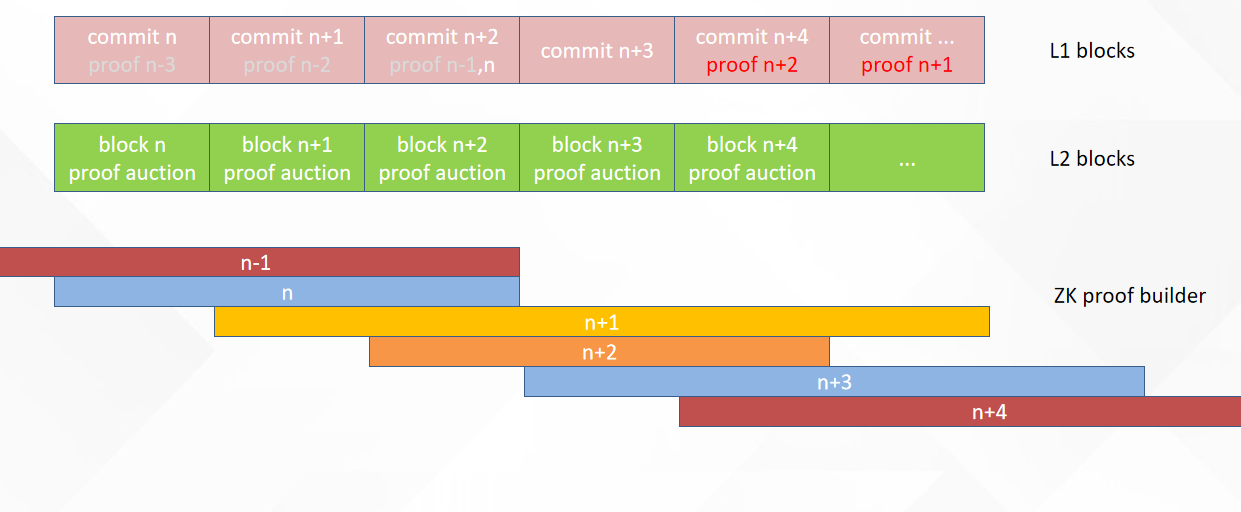
**Risks**:

Proposer may send tx and add it to CRList to get MEV.

Builders’ll try to get MEV, but its block may not be choosen.But anyway, one builder’s block will be choosen,we can’t avoid it.

If roll back happened on L2, ZK-proof market should be adjusted.

### 5.3 Stage 2&3



On above figure, we have L1 & L2 blocks, and the ZK proof builders. Actually L2 blocks maybe much more than L1, here’s just a schematic diagram to make it easy. 1st we have blocks reached consensus on L2, e.x. block n, n+1, n+2...etc, after each block confirmed on L2, we’ll commit the details on L1, that’s a promise that should be proven later, it’s commit n, commit n+1, n+2...etc.

We have many candidate ZK proof builders, and an auction of the right of building a proof for certain block. When block n is confirmed on L2, the auction of block2 starts, the ZK builders needs to evaluate and promise how long it can create the ZK proof, the auction contract will consider the promise, and the weight that builder has, to decide which ZK builder gets this job. The builder’s weight maybe related to stake, and how he worked in the history, e.x. if he exceeded the time he promised before, or he uploaded the false proof, the weight maybe decreased.We can see the blue builder get the right to build proof for n, after he finished it, he take the auction again, and get n+3.

Here we can see some interesting things, different ZK builder has different machines, so their time of finishing a proof is different. Here in the L1 block with commit n+2, 2 proofs are supplied in the same block. And let’s assume orange builder generates proof faster than the green one, so proof n+2 is supplied earlier than n+1. But that’s ok, because it won’t have influence to the L2 consensus, it only has influences on L1 confirm time. But if roll back happened on L2, it’ll be complex. I can research on it if necessary.

## 6. Should we use DAS?

### 6.1 DAS target and cost

Celestia uses DAS to ensure light client security, and make the chain scalable. Light clients won’t download the entire block, it only downloads the block head, so it maybe cheated by malicious nodes. Malicious nodes may send a fake block head to light client, and keep some part of the block, so honest full node can’t generate fraud proof for the light client. Celestia’s solution is to use DAS to make sure all block data is available.

But DAS has big cost:

1. Block size \*4 if use 2D RS-Code.
2. KZG proof
3. Light clients sample the block data

### 6.2 Alternative solution for protect light client

If our target is to protect light clients, we have alternative solutions. Here’s one:

**Assumptions**:

* all nodes, including light client or full client, could obtain the right genesis block.
* any light client could connect to at least 1 honest full node.
* most validators are honest (at least 2/3)

**Validators** :

* In the genesis block, all initial validators are decleared.
* whenever validators are changed, the new validator set should be updated in the block head, or at least a proof should be at the head, so light clients could get the right validator set.
* When a new block is produced, validators not only sign the block, but also sign the head.

**Light clients**:

* At any time, they can get the latest, and right validator set, and stores it. (This can be guaranteed by above actions).
* Only accept the new block head that’s signed by the right validator set (At least 2/3 validators signed it), they can verify the signature , they can drop the signatures to save resource.

By doing above job, light clients also can get a right block head list, and the cost is low (only save validator list, and verify some signatures) . And we don't need DAS . Actually similar solution is used in COSMOS for cross chain.

### 6.3 Trade off

Comparing to the easy solution, DAS has its advantages, here it is:

* Easy solution only works when at least 2/3 validators are honest. Although Celestia needs at least 2/3 honest validators too. But if not, the damanage that malicious validators can cause , is much less than easy solution.
* In DAS solution ,they could only roll back the chain,double spend. but they can't create invalid tx (like create unlimited token). if only 1 honest full node exists, he can send fraud proof.
* In easy solution, the malicious validators can send invalid transaction, so they can make unlimited tokens, etc. and light clients won't find it, because they can hide the invalid transaction, so honest full node can't create fraud proof.
* Celestia is a sharding system, so different shardings work like light clients when there’s a cross sharding transaction, in DAS solution, they may receive fraud proof from other shardings.
* If some block data are lost, full node can rebuild the block on the help of light clients in DAS case.

So we should choose proper solution on certain requirements.

## 7. Private key recovery

### 7.1 target

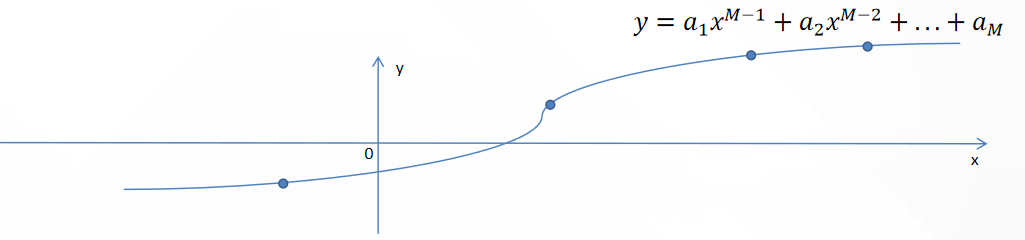
Game players are used to use username/password to login to the game, but blockchain uses sk/pk/address, it’s important to give players the web2 user experience to bring them into blockchain world.

We should consider below targets:

1. Generate key pairs for players, and use it on chain.
2. When they’re switching to new device, we should keep the key pairs with their login action.
3. When player forgot his/her password, we should have a way to recovery it.
4. When player try to play different games, we should have ability to guide him/her to use the same address.

Using keystore, it’s easy to achieve 1,2. for 4, let’s consider it later. I wish to handle 3 here. Some chain/dapp can update private key for the account, but it can’t be used on Ethereum, and it break the Deterministic association between pk and account, it’ll make dapp design complicated (ex. Needs an query to make sure a pk is used for an account).

### 7.2 solution



We still use keystore, we let user set N recovery questions, if user could answer M (M≦N) questions, we could recover the original sk. M should be at lease 2.

1. Split the private key into M pieces, we can get a polynomial whose order is M-1, the private key pieces are the coefficients .
2. Convert the x (M≦x≦N) answers of recovery questions to AES keys, randomly choose x points on the polynomial, use the AES keys to encrypt them. write the recovery questions and ciper text into the keystore.
3. When recoverying, we could convert the answers to AES keys, then decrypt the x points, then rebuild the polynomial, then we get the original private key.

We could regenerate the address from recovered private key, to compare with the original stored address, to check whether recovery succeed. The solution has below advantages:

1. It recovers the original sk, so it can be used for Ethereum. It keeps the deterministic association between pk and account.
2. Don’t need anyone’s help. Neither the social relationship, nor the dapp service. It’s operated locally.
3. It’s common, all dapps could use this solution.

## 8.Q&A

**Q : When does the tx should be treated as finished, can be trusted.**

A : We have many time points can be candidate for it.

1. The L2 block was produced.
2. At this time, txs had been put into a block, but the L2 sequencers don’t have the state of the L3 DApps, so some txs maybe marked invalid later (in the receipt file). So this point isn’t proper.
3. The L3 DApp node executed the txs in block, and committed the post-state to the “State root contract”, and published the receipt doc to DA. At this time , the latest state is fixed, and we know whether a tx is valid. I think this’s a proper point for normal users think the tx is finished.
4. Here we have 2 possible malicious actions:
   1. The block is reorged. It can be found and make a fraud proof. Because the sequencers needs to publish the block to DA, and sign it. So if we found 2 blocks are on the same height, with sufficient signatures, we can make a fraud proof and the sequencers signed on both of the blocks should be slashed. That’s at least 1/3 of it.
   2. When committing to L1, your tx is not included, or your state is not updated. In this case, fraud proof is easy to make ,sequencers will be slashed.

So for normal game use, this point is ok. And normally it should be within 3 seconds after the tx is sent.

1. States had been published on L1.
2. Although Ethereum maybe reorged too, but I think this point is good enough for almost all DApps. But it may take 13 seconds or even longer.

**Q : How to make sure the L3 nodes work properly.**

A : I think this’s the most important problem. The best way is to use ZK EVM. But before it’s practical and cheap enough, we should consider an alternative solution.

First of all, L3 is not decentralized. The main point here is when L3 commit state root to L2 contract, the root must be correct. This could be confirmed by Verifier.

Verifier could be a module of sequencer. When a sequencer built a block, it has responsibility to verify the state transit file, and confirm on the state contract.

The left work is how could L3 get trusted by its user. This should be considered by the L3 itself, not by the L2. here’re some analyzes :

I think on L3, there’re 3 kinds of DApps.

1. Normal DApps, e.x. a game. They don’t want to run the nodes, they just want to find a chain that secure ,fast and cheap enough to run their contracts.
2. In this case , we need to deploy and run some L3 by ourselves, like the bootstrap, to supply examples for the community how to build L3 . normal DApps can directly run on it. And nodes of this L3 should stage for security, like EigenLayer can be used.
3. Some other companies or groups wish to build a L3 on our L2. They should consider it by itself.
4. Big DApps, e.x. Uniswap. They want to make sure their txs can be properly handled, they may wish it run on a separate chain. On this case, they may wish to build their own chain. Operating their own chain. For this kind, they may consider how to make it secure enough for themselves. Maybe the EigenLayer, maybe not. We don’t need to worry about it too much.

**Q: The bottleneck of performance of this system is L2 orderer, how to improve its performance?**

A: I think the case here is exactly the same with Celestia.

L2 sequencers should be the bottleneck here. They need to do 4 things:

1. Check the signatures
2. Order the txs
3. Get Consensus.
4. Write block to DA.

1，2，4 are easy to handle. 3 may cost time, because the block size maybe large, so bandwidth requirements maybe high. We can do some work to improve it:

We may just have the tx hash in the block during the consensus if we could make sure the tx data is available.maybe we could rebuild the whole block locally by the sequencer, or we could use prove of available. I saw some chains do it this way.