PoE(Proof of Events) Whitepaper

A Highly Scalable Protocol for high throughput and low latency Transaction processing

[Technical whitepaper — release 2 — revision 2]

June, 2023

By Larry Liu

http://roturalabs.com

Abstract

This paper describes the design of Proof of Events mentioned in Rotura Protocol whitepaper. The PoE is responsible for forming agreement on ordering and finalifizing transactions among billions of users.

1. Introduction

Blockchain technology needs breakthrough. But present blockchain architectures all suffer from a number of issues not least practical means of performance and scalability, which prevents billions of users from adopting cryptos.

The focus of this paper is to formalize the scale out architecture based on new sharding solution, and a new streaming protocol to increases throughput without compromising security and decentralization.

2. Define the challenges

Much of Ethereum's success can be attributed to the introduction of its decentralized applications layer through EVM, Solidity and Web3j. While Dapps have been one of the core features of ethereum, scalability has proved a pressing limitation. Considerable research has been put into solving this problem, however results have been negligible up to this point.

Sharding was first used in databases and is a method for distributing data across multiple machines. This scaling technique can be used in blockchains to partition states and transaction processing, so that each node would process only a fraction of all transactions in parallel with other nodes. As long as there is a sufficient number of nodes verifying each transaction so that the system maintains high reliability and security, then splitting a blockchain into shards will allow it to process many transactions in parallel, and thus greatly improving transaction throughput and efficiency. Sharding promises to

increase the throughput as the validator network expands, a property that is referred to as horizontal scaling.

There are solutions of the scalability including different sharding solutions such as Omniledger, Zilliqa and Algorand etc. Apparently, those sharding approaches doens't solve the scalability problems. The sharding solution should be designed for the computation instead.

3. Design Principle

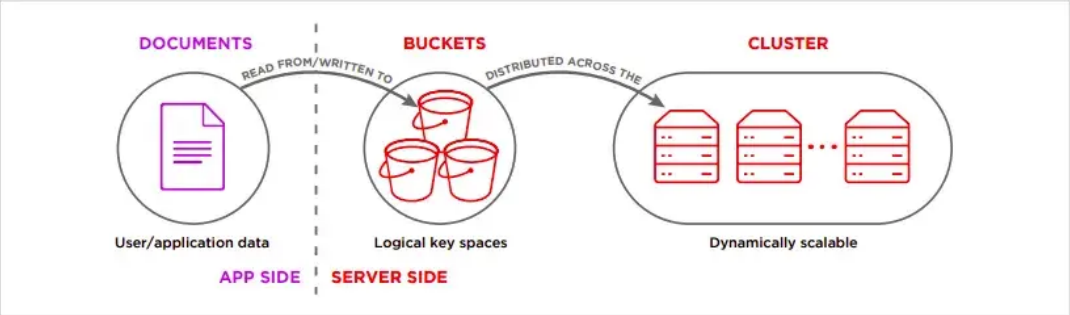
As a distributed system, Sharding design should consider more for the distributed computing. Our research shows that the sharding plan should be able to scale with the computation. And it has to be able to handle growing number of transactions. Meanwhile, it has to work with the validators to handle those transactions. Rotura protocol whitepaper introduces a new approach of scale out blockchain. The design of the PoE is to work with PoDAG to build a new architecture for the future blockchain.

4. Proof of Events

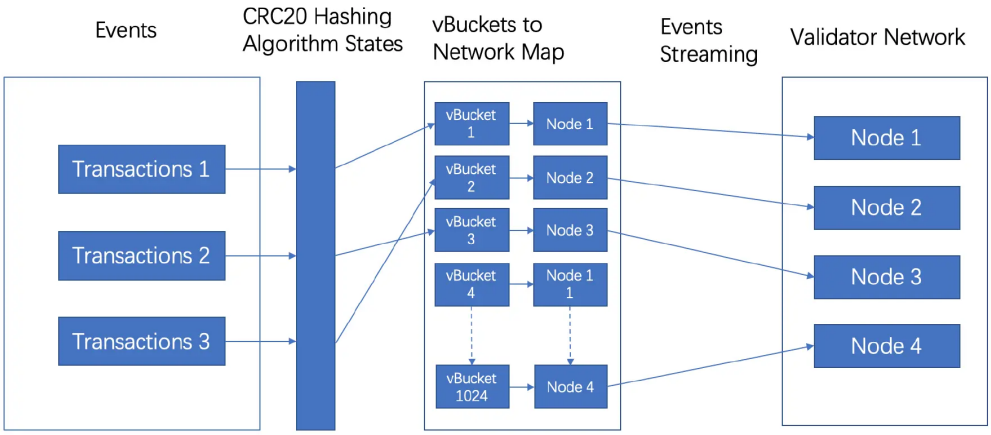
Proof of Events is a sequence of computation that can provide a way to cryptographically verify passage of sequence between two events. It uses a cryptographically secure function written so that output cannot be predicted from the input, and must be completely executed to generate the output. The function is run in a sequence on a single core, its previous output as the

4.1 PoE Automatic Sharding

Sharding is a popular technique for database performance optimization. When data size becomes bigger, it becomes difficult to place it in a single machine, performance & scalability takes a hit after some level of vertical scaling, cost sky rockets. This is especially true for blockchain that serve huge scale of users, store huge amount of data.



A bucket is again divided into `1024` virtual buckets (`vBucket`). Each virtual bucket is assigned to a physical node as shown below. Each `vBucket` acts as a logical shard. `vBucket` to node mapping is handled by validator cluster manager automatically.



Every transaction is assigned an id / key by the concerned application. The client creates a `CRC32` hash of the key which acts as a shard key. The `hash % 1024` determines a number indicating which `vBucket` contains / should contain the data.

Once, the `vBucket` is identified, from the `vBucket` to node mapping, the physical node is identified and the actual document is written to / fetched from there.
Thus, `vBucket` actually divides a bucket into smaller chunks. Since, a `vBucket` is automatically assigned to a physical machine based on different parameters, each physical node almost homogeneously shares the data set, thus making the system more scalable, high–performing and reliable.

What is the benefit of using vBucket?
Deterministic hash functions always map the same data to the same `vBucket` for its life time as number of `vBucket` is constant per bucket. Thus the data to `vBucket` mapping remains constant from a client's point of view but the cluster has full flexibility to move the `vBucket` around when a new node is added, removed or some cluster change event happens. This makes the **physical layer of the system decoupled from the logical layer**.

Also, even if the individual buckets are extremely skewed in terms of number of events contained, `vBucket` roughly equally share the burden of the records, thus making the distribution better, scalability and performance of the system predictable.

4.2 PoE Stream Protocol

PoE uses DCP protocol that significantly reduces latency for transaction verification. With DCP, transactions in memory are immediately streamed to be verified without being written to disk.

4.2.1 The Database Change Protocol

The Database Change Protocol (DCP) is a data synchronization protocol. It provides features and concepts that allow the development of rich and interesting features. DCP can be viewed as the heart of PoE because it pumps data out to other components in the system so that they can function properly.

DCP was developed from the ground up to quickly and efficiently move massive amounts of data around the cluster and datacenter. It provides guarentees of consistency so that applications that consume data with the DCP protocol can make sure that they are making decisions based on a consistent view of the database. DCP also provides fine grained restartability in order to allow consumers of DCP to resume from exactly where they left off no matter how long they have been disconnected. Fine grained restartability is crucial when dealing with massive amounts of data because retrasmitting data that has already been sent may impose high costs. Finally DCP provides high throughput, low latency data transfer to allow applications to get the lastest data change as quickly as possible.

DCP is built on four major architectural concepts that allow the protocol to be used to build a rich set of applications. Below are the major concepts.

Order

The first major concept is order, and it is the foundation for the rest of the major architectural concepts. Order is important because it allows applications to reason about causality of data – or in other words allows an application to know if an operation occurred before or after another operation. DCP achieves order through the use of sequence numbers, Snapshots and failover logs. Sequence numbers are used to keep order on a single node and failover logs are used to resolve ordering conflicts during failure scenarios.

Restartability

Restartability is an important feature for any replication protocol that often has to deal with large amounts of data. In any distributed system components crash and when you're moving large portions of data applications want to be able to resume from exactly where they left off and not have to resend any data in the case of a dropped connection. Due to the DCPs ordering mechanism restartability is possible from any point and means the server won't send any data that the application has already received even if the application has been disconnected for an extended period of time.

Consistency

Some applications can only make decisions about the data they receive if they have a consistent view of the data in the database. DCP provides consistency through snapshots and allow applications know that they have seen all mutations in the database up to a certain sequence number.

Performance

DCP provides high throughput and low latency by keeping the most recent data that needs to be replicated in memory. This means that DCP connections will normally not have to read data off of disk in steady–state.

4.2.1.1 Snapshots

Snapshots are very critical to achieve the above 4 architectural foundations of DCP.
In the most basic sense, a snapshot (a full snapshot) is an immutable view of the KV store at an instance. This is also a consistent view of the KV store at that instance.

In PoE, Snapshot is an immutable copy of the key–value pairs in a vbucket (partition), received in a duration of time. That duration of time is marked by a 'start sequence number' and an 'end sequence number'. Successive snapshots give a consistent view of the vbucket up to a particular point, that is, till the 'end sequence number' of the last snapshot. Snapshots provide restartability when streaming items to replicas or other DCP clients. Snapshots also deduplicate multiple versions of the key by keeping only the latest value of the key.

There are two types of snapshots that are formed when streaming items from the vbuckets. When a client connects to the source, it initially gets a 'disk snapshot', and later when the client catches up with the source and hence has reached the steady state, it starts getting 'point–in–time' snapshots from memory.

Disk Snapshots

Disk snapshots are immutable persistent snapshots on the disk. They are formed after a batch of items are written onto disk. These snapshots persist on the disk until multiple such snapshots are compacted into a single snapshot. The replication clients then pick up these immutable snapshots and hence get a view of the vbucket that is consistent with the source vbucket at the time of starting the stream. This is called a disk backill. When a disk backfill starts, all the snapshots are logically merged and sent over as a single snapshot to the client.
For example, say the disk has 3 snapshots:

- `snp1` from 1 to 20
- `snp2` from 21 to 30
- `snp3` from 31 to 60.

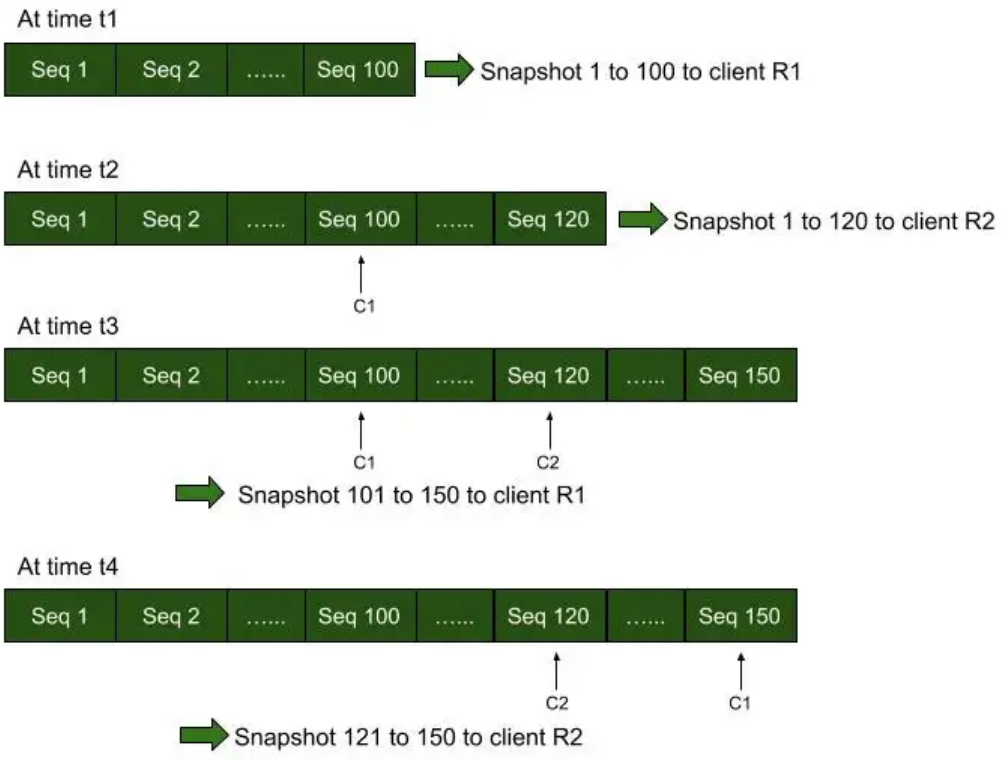
A backfill request will merge all 3 snapshots and send over a single logical snapshot from 1 to 60. For request from the middle of the snapshot, that if the request is from sequence number 15, then the logical snapshot sent over is 15 to 60. Note that, to get a consistent view of the vbucket, the client should read till the end of the snapshot as some of the keys might have been de–duplicated.

A drawback of the disk snapshots is that the keys cannot be replicated until the snapshot is formed. They are good when the snapshots are fairly large, that is when replication clients picks up batch of items and when they are fine with the higher latency in being synced with the source.

Point–in–time snapshots (in–memory snapshots)

Point–in–time snapshots are the snapshots that are created on the fly. That is, the source vbucket creates the snapshot only at the time when a replication client requests data. In PoE, point–in–time snapshots are in–memory snapshots created by the checkpoint manager. They are best suited for steady state replication of items from memory. Steady state means all replication clients have almost caught with the source vbucket.

Below is an example how point–in–time snapshots give consistent view of the vbucket across multiple clients:



1. Say the source vbucket has items from sequence number 0 to 100 and a replication client `R1` makes a request for the copy of data. Items from sequence number 0 to 100 are sent as a snapshot (point-in-time snapshot) to `R1` and a cursor `C1` that corresponds to the client `R1` is marked on the checkpoint manager.
2. At a later time, say 20 more items are appended and hence the highest sequence number is 120. If another client `R2` requests for data, items from sequence number 0 to 120 are sent as a snapshot to `R2` and a cursor `C2` that corresponds to the client `R2` is marked on the checkpoint manager.
3. When more data is appended to the vbucket, say until sequence number of 150, the client R1 can get up to 150 in a successive point-in-time snapshot from 101 to 150.
4. The client `R2` can get up to 150 in a successive point-in-time snapshot from 121 to 150.

Note that cursors `C1` and `C2` are important to start over quickly from where the clients `R1` and `R2` had left before. As the cursors move, the key-value pairs with sequence numbers less than the sequence number where any cursor is marked can be removed from memory.

Point-in-time snapshots are good for steady state replication as the replication clients can get their own snapshots that are created on demand and hence avoiding any wait for a snapshot to be created, thereby catching up with the source very fast with the latest key-value pairs shipped over as soon as possible.

Slow clients and lagging (deferred) clients do not work well with point-in-time snapshots, as they force the checkpoint manager (memory) to keep alive Checkpoints which have Cursors referencing them, so the slow clients can increase the memory usage. We handle such slow clients by dropping the respective cursors on the checkpoint manager and falling back to disk snapshots.

4.2.1.2 Deduplication

Deduplication is the removal of duplicate versions of the same key in a snapshot and retention of only the final version of the key in that snapshot.

In a bucket, deduplication is done both in memory and on disks. In memory, deduplication is done when an item update is written onto an open (mutable) checkpoint where an item of the same key already exists. On disk, deduplication is done when multiple disk snapshots are compacted into a single disk snapshot during compaction. However, when multiple disk snapshots are merged logically into a single DCP backfill snapshot deduplication is not done.

In an Ephemeral bucket, deduplication is done periodically by removing stale copies and also when a backfill snapshot is formed.

Why deduplicate?

The motivation for deduplication is to reduce the amount of space needed to store documents on disk, and to reduce network bandwidth when replicating changes via DCP. Without *any* deduplication, a bucket would need to keep *every* copy of every mutation ever performed, which quickly becomes an unmanageable amount of memory and disk. However, the presence of deduplication means that care must be taken when DCP clients process a snapshot marker, as until a complete snapshot has been processed the state is inconsistent.
For example, consider the case where a vbucket creates document A, creates document B, then modifies document A again. Due to deduplication, the checkpoint manager will discard seqno 1 (it has been superseded by seqno 3) and the in-memory state will look as follows:

```
Seqno   [1]  2   3
Key      A   B   A
```

`[N]` = *mutation has been deduplicated*

If a DCP client requests to stream from this vbucket (starting at the beginning) they will be sent:

- A SnapshotMarker specifying the extent of the mutations to be sent: `start=2` and `end=3`
- A Mutation for `2:B`
- A Mutation for `3:A`

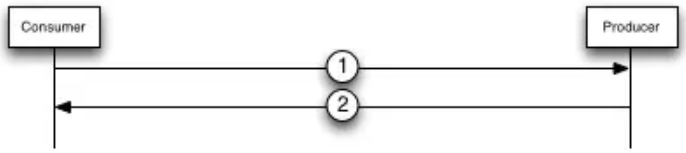
If the DCP client receives and processes all three messages immediately, all is well – they know about all documents in the bucket (`A` and `B`), and they have the latest version of each one.
However, consider what happens if the DCP client only received the first two messages – they only know about document B. If they exposed this state to an application as-is, it would be inconsistent – there never existed a point in time when document B existed and A did not. To ensure consistency the DCP client must not publish a partial snapshot – they must buffer of otherwise hide such state until the last seqno of the snapshot has been processed.

4.2.2 Protocol Flow

This document intends to describe an example of the protocol flow between a Consumer and a Producer. The sections below will provide a step-by-step walk through of the messages exchanged by the Consumer and Producer for the lifetime of a connection.

4.2.2.1 Creating the Connection

Creating a connection consists of opening a TCP/IP socket between the Consumer and the Producer and sending an Open Connection message. This Open Connection message message signifies this connection is a DCP connection and also allows the sender to give the connection a name. Below is a diagram of the interaction between the Producer and Consumer that takes place to establish a DCP connection.

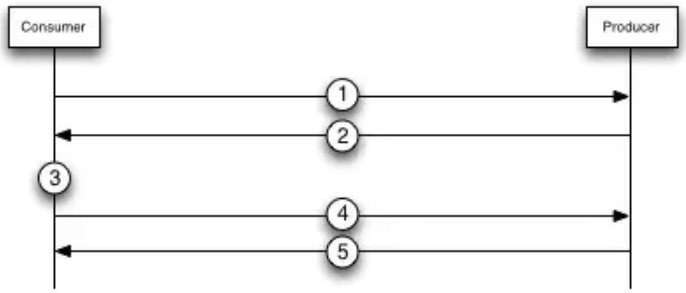


1. The Consumer will send an Open Connection message to the Producer in order to create the initial connection and assign the connection a name. The connection name can be used to get statistics about the connection state as well as other useful debugging information. If a connection already exists on the Producer with the same name then the old connection is closed and a new one is opened.
2. The Producer will respond with an *OK* message to signify that the connection was created.

Once a DCP connection is created the Consumer can configure the connection by sending Control messages to the Producer. Control messages can be used to do things like enable dead connection detection and flow control. Sending Control is not required if the default connection settings are sufficient.

4.2.2.2 Starting the Stream

Once a connection has been created the Consumer can to create one or more VBucket Streams in order to stream data from the network. The diagram below describes how to create a VBucket stream once a connection exists.



The Consumer will send a Stream Request message to the Producer. This message indicates that the Consumer wishes to receive data from the Producer for a specific VBucket. If this is the first time the Consumer is connecting to the Producer then 0 should be specified in the *Start Sequence Number*, *VBucket UUID*, *Snapshot Start Sequence Nunmber*, and *Snapshot End Sequence Number* fields to indicate that the

Consumer wants to receive all data from the network from the beginning of time. If it is not the first time the Consumer is connecting to the Producer then the Consumer should specify the last sequence number the Consumer received in the *Start Sequence Number* field, the most recent VBucket UUID in the Consumers failover log int the *VBucket UUID* field, and the most recent snapshot information for the *Snapshot Start Sequence Number* and *Snapshot End Sequence Number* fields. This will allow the Consumer to begin receiving data from where it last left off.

1. The Producer will respond to a Stream Request with either an *Ok* message or a *Rollback* message. If the message contains a *Rollback* response then the stream behavior will resume at step 2, but if an *Ok* message is received then jump to step 5.
2. The Producer sends a Rollback response which contains a *Sequence Number* that needs to be rolled back to. This means that the Consumer and the Producer have different histories of their data and the Consumer needs to remove some of its data that the Producer doesn't have and then try to start the stream again.
3. What happens in this step depends on the application that the Consumer is streaming data for. Some applications might want to rollback some of their data and other might not care.

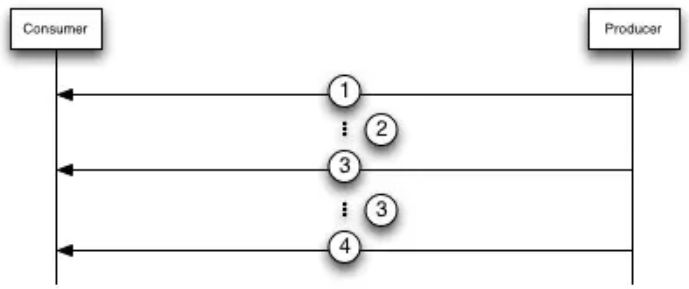
The Application then uses its new last Sequence Number and VBucket UUID and sends another Stream Request message to the Producer. The Sequence Number should be less than or equal to the Sequence number received in the Rollback message.

1. It is possible if there is a failover between steps 3 and 4 that the consumer will need to rollback again. This should be rare, but if it does happen then jump back to step 2.
2. The Producer sends an *Ok* message which contains the Failover Log back to the Consumer. The Consumer should persist the Failover Log (and use the latest entry for the next request) and expect to begin receiving data from the Producer.

Starting VBucket Streams can be parallelized since the Consumer can start as many streams as it needs to. Consumers should not create a connection per VBucket Stream since this will cause heavy resource usage on the server since each connection uses a file descriptor.

4.2.2.3 Reading the Stream

Once a stream is started the Consumer will begin receiving data. The figure below shows what messages the Consumer should expect to receive. Fully understanding this section requires understanding what a snapshot is and why snapshots are significant.



1. The first message that will be sent during a stream request is a Snapshot Marker. The purpose of this message is to tell the Consumer the sequence numbers that are part of the snapshot that is going to be sent by the Producer. The *Snapshot Start*

Sequence Number and *Snapshot End Sequence Number* in this message should be persisted since they are required to properly restart the stream if the connection is lost.

After sending a Snapshot Marker the Producer will send a series of Mutation, Deletion, and Expiration messages that are part of the current snapshot.

1. Once all of the items from the first snapshot have been sent the Producer will check to see if the sequence number of the last item sent is greater than the *End Sequence Number*. If it is then the stream will be closed. If not then more items will be sent.
2. If there are more items to send then the Producer will send another Snapshot Marker followed by more Mutation, Deletion, and Expiration messages.
3. At some point however the VBucket stream will be finished and the Producer will send a Stream End message to signify the stream is finished. A stream may end for different reasons so it is important to check the status code in the Stream End message.

4.2.2.4 Closing a Connection

When the Consumer wants to end a DCP connection it should simply close the socket. The Producer will see the connection going down and this will cause the Producer to remove any structures used for sending data to the Consumer.

5. Conclusion

Proof of Events protocol is the digital equivalent of the human body's central nervous system. It is the technological foundation for the 'always-on' blockchain world where businesses are increasingly software-defined and automated, and where the user of software is more software.

Technically speaking, PoE is the decentralized technology of capturing transactions in real-time in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. PoE thus ensures a continuous flow and interpretation of transactions so that the right information is at the right place, at the right time.

What else can I use Proof of Events for?

Proof of Events is not only designed for high scalable blockchain, it can also be applied to a wide variety of use cases across a plethora of industries and organizations. Its many examples include:

- To process payments and financial transactions in nearly real-time
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.

- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.
- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.