Sawtooth Lake 0.8
State Delta Subscription

DRAFT

Overview

The goal of State Delta Subscription is to provide a mechanism for exporting on-chain state values from a validator to an external data store. This allows applications to efficiently query their state values in cases where there are complex relationships represented in their data. This efficient off-chain state access comes at the expense of relying on a single validator for state updates, this puts the application at risk of having stale data or forked state if the validator supplying the state updates comes out of consensus or is disconnected from the network.

This design provides a combination of reference implementation and schema recommendations for implementers.

Design

Subscriptions involve both a validator and a client framework to work in concert to stream state changes as they are created via the process of block validation and publishing.

The validator will collect state deltas, keep them in an out-of-band store, and send the information to registered client subscribers.

Clients subscribe to a validator for state deltas on a specific subset of namespaces. The client will store them in a database, which can be used for richer queries and represent relationships with application-specific off-chain data.

Execution

Validator

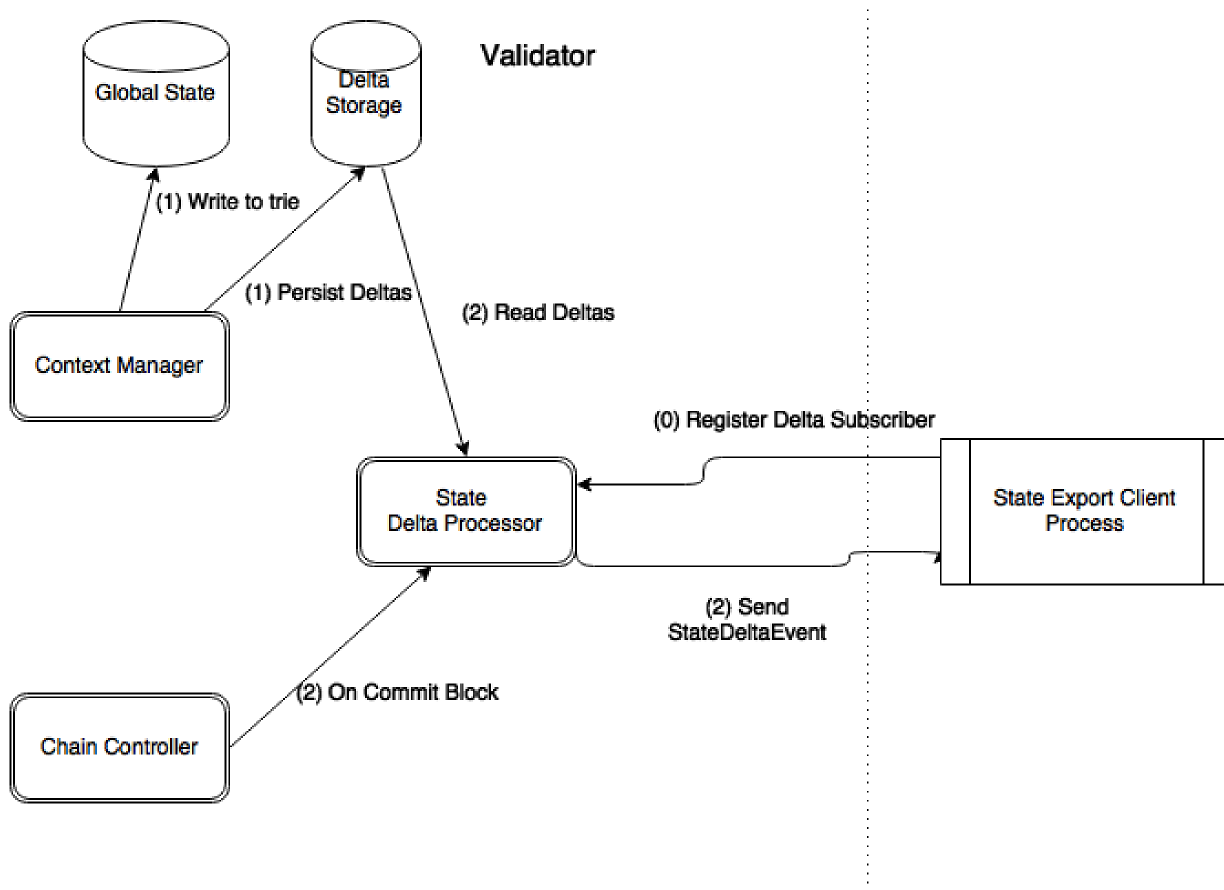

Figure: Validator Delta Processor

The validator listens for messages for registering and unregistering state event subscribers. These subscribers are managed by the `StateDeltaProcessor` (step 0 in the above diagram).

The validator collects and stores the deltas within the `ContextManager` (step 1). The ContextManager collects the deltas at the time of a context squash and stores them on disk, locally to a validator. This information is out-of-band (i.e. it is not guaranteed to be the same for each validator in a network), but computed only via changes to blockchain. The storage should be a mapping between a state root hash and a collection of `StateChange` objects.

When a block is committed (step 2), the state deltas for the block's state root hash will be retrieved via the `StateDeltaProcessor` and sent to subscribers with the block's header signature (i.e. block id) and block number.

State Delta Subscription Client

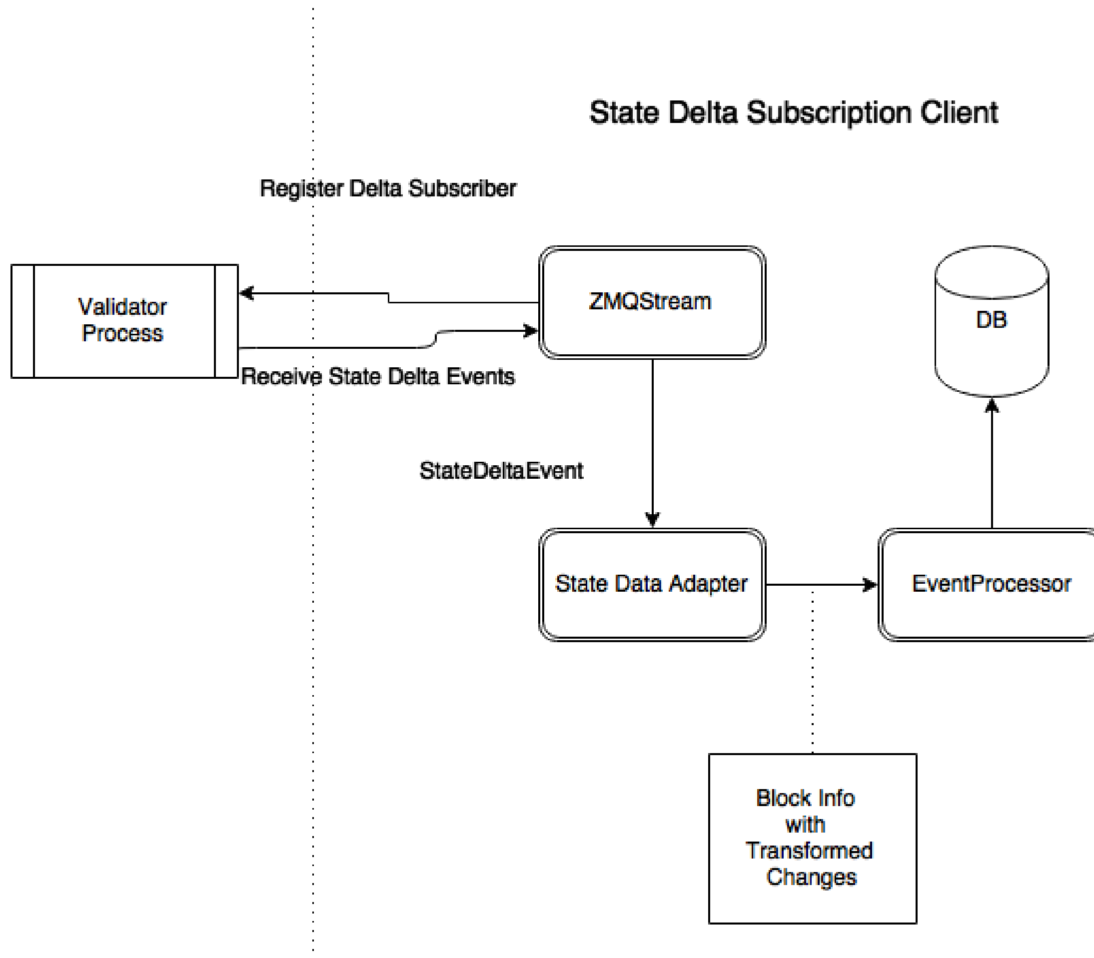

Figure: Subscription Client

A state subscription client operates in the in the following way: First, it connects to a validators ZMQ interconnect endpoint, and sends a `RegisterStateDeltaSubscriberRequest` message. This registration should include the most recent block id known to the client, and the address prefixes of interest (that is, it will only receive state values that match the prefixes). It should then be ready to listen for `StateDeltaEvent` messages, which should begin from the next block from the one given.

When the client receives a `StateDeltaEvent`, it first transforms the changes from raw state values to domain-specific values via a `StateDataAdapter`. The client ensures that the new block does not represent a fork, and if so updates the database accordingly (see the Fork Resolution section). Existing records for the domain-specific objects are updated as “ended” and new records are inserted as “started” with the new blocks block number. See the section on Storage Schemas for more information on modeling tables.

If the client received the status UNKNOWN_BLOCK when registering itself as a subscriber, this most likely is the result of a fork having occurred and been resolved with the validator node. The client can send an increasing sized set of known block ids, until it returns a valid registration. For example, it can send the previous five ids; if that fails the previous ten, and so on. Once it receives a status of OK, it will receive event messages as normal, handling the fork resolution as per normal operation.

The following pseudocode demonstrate the handling of an event.

```
on_receive_event(state_delta_event):
  begin transaction:
    block_id := state_delta_event.block_id
    block_num := state_delta_event.block_num
    state_root_hash := state_delta_event.state_root_hash
    changes := state_delta.changes
    domain_entries := data_adapter.transform_state_entries(changes)

    existing_block := query_block_by_block_num(block_num)

    if existing_block:
      resolve_fork(existing_block) # See below

    insert_block(block_id, block_num, state_root_hash)
    for domain_entry in domain_entries:
      update_end_block_num_table_for_type(domain_entry, block_num)
      if change.operation = SET:
        insert_new_entry_table_for_type(domain_entry, block_num)
   commit transaction
```

Fork Resolution

State clients must deal with the problem of fork resolution. The table structures as recommended, coupled with the behavior of the validator as it processes blocks, make this process relatively painless.

The following pseudocode demonstrates the fork resolution process:

```
resolve_fork(existing_block):
  for table in domain_tables:
    delete from table where start_block_num >= existing_block.block_num
    update table set end_block_num = null \
       where end_block_num >= existing_block.block_num
  delete from block where block_num >= existing_block.block_num
```

Interconnect Messages

The following protobuf messages will be needed:

```
// Registers a subscriber for StateDeltaEvent objects.  The
// identity of the subscriber will be based on the ZMQ connection
// id. This is an idempotent request.
message RegisterStateDeltaSubscriberRequest {
    // The block id (or ids, if trying to walk back a fork) the
    // subscriber last received deltas on.  It can be set to empty
    // if it has not yet received the genesis block.
    repeated string last_known_block_ids = 1;
    // The list of address prefixes of interest.  Only state changes
    // that occur on values in the given prefixes will be sent to the
    // subscriber.
    repeated string address_prefixes = 2;
}

// The response to a RegisterStateDeltaSubscriberRequest
message RegisterStateDeltaSubscriberResponse {
    enum Status {
        // returned on successful registration
        OK = 0;
        // returned on a failed registration, due to
        // an internal validator error
        INTERNAL_ERROR = 1;
        // returned on a failed registration, due to the
        // last_known_block_id being unknown.  This could imply
        // that a fork had occurred and been resolved since
        // last unregistration.
        UNKNOWN_BLOCK = 2;
    }

    Status status = 1;
}
```

```
// Unregisters a subscriber for StateDeltaEvent objects.  The
// identity of the subscriber will be based on the ZMQ connection
// id.  This is an idempotent request.
message UnregisterStateDeltaSubscriberRequest {
    // No data
}

message UnregisterStateDeltaSubscriberResponse {
    enum Status {
        // returned on successful registration
        OK = 0;
        // returned on a failed registration, due to
        // an internal validator error
        INTERNAL_ERROR = 1;
    }
    Status status = 1;
}

// A state change is an entry in a given delta set. StateChange objects
// have the type of SET, which is either an insert or update, or
// DELETE.  Items marked as a DELETE will have no byte value.
message StateChange {
    enum Type {
        SET = 0;
        DELETE = 1;
    }
    string address = 1;
    bytes value = 2;
    Type type = 3;
}

// A StateDeltaEvent contains the information about the start and
// end of the delta (from a block perspective) and the list of
// changes that have occurred in that time. The list of state
// changes are limited to those in the namespaces specified at
// subscriber registration time.
message StateDeltaEvent {
    string block_id = 1;
    int32 block_num = 2;
    string state_root_hash = 3;
    repeated StateChange state_changes = 4;
}
```

On initialization, a subscriber can query the validator for the current block and state contents using the existing client messages (e.g. using a `ClientStateListRequest`) to prepopulate the database tables before registering a state delta subscriber. This will allow a client to connect to a long-running validator and catch up without needing to operate on a delta-by-delta basis.

State Data Adapter

A State Delta Subscription client needs to translate the opaque bytes from state values it receives in a `StateDeltaEvent` to application-specific values, which can be stored in the database. This can be handled by the application developer by implementing a `StateDataAdapter`. One adapter is needed for each address prefix of interest expressed during subscriber registration. These adapters are registered with the `Materializer`, used to transform the state changes. Any state data that arrives that doesn't have an `StateDataAdapter` will be dropped.

While the implementation of the individual `StateDataAdapter` implementations are domain specific, the implementation of the `Materializer` collecting module should be provided by sawtooth-core.

The following pseudocode shows the required protocols for these adapters:

```
protocol StateDataAdapter:
    transform(state_change: StateChange): DomainObject

protocol Materializer:
    register_namespace_adapter(address_prefix: string,
                               adapter: StateDataAdapter)

    transform_state_entries(state_changes: [StateChange])
```

Storage Schemas

A State Delta Subscription client has three types of tables: sawtooth-core tables (namely block, described below), domain-specific state tables (tables populated using the results of the `Materializer`), and application tables (tables supporting off-chain application components).

All clients should include a table for block history. This is needed to handle fork resolution. It should have the schema as the following (this example is given for a PostgreSQL database, but can be adapted to others):

```
CREATE TABLE block (
    block_id varchar(128) CONSTRAINT pk_block_id PRIMARY KEY,
    block_num integer,
    state_root_hash varchar(64)
);
```

Storage schemas for a State Delta Subscription client define a pattern for managing the data for a particular application in a state table. These tables should follow the guidelines of [Type 2 Slowly Changing Dimensions]. For the use in a State Delta Subscription, a state entry table row should include additional columns of `start_block_num` and `end_block_num`, with an sequence id column (or another unique identifier scheme) as the primary key. This id column removes any constraints on the resulting domain data. These columns specify the range in which that state value is set, or exists. Current values (i.e. those that are valid as of the current block) have `end_block_id` and `end_block_num` set to `NULL` (or MAX_INTEGER, if the database doesn't support null fields in an efficient way). For better query performance, it is recommended to create indexes on the natural key of the entry and the `end_block_num`.

Take the Intkey transaction family as an example. The state fields for each entry have a string name and integer value. The table for the state entries of this family would look as follows in PostgreSQL:

```
CREATE TABLE integer_key (
    id BIGSERIAL CONSTRAINT integer_key_pk PRIMARY_KEY,
    intkey_name varchar(256),
    intkey_value integer,
    start_block_num integer,
    end_block_num integer,
);
```

```
CREATE INDEX integer_key_key_block_num_idx
  ON integer_keys (intkey_name, end_block_num NULLS FIRST);
```

Off-chain application tables should not reference entries in these tables directly, via a foreign key, as that only tracks a state value for a given block in the chain. Instead, references to a state table from an application table should be by natural key only. For example, using an Intkey example, the application table would reference the column `intkey_name`. The data would then be joined on the end_block_num field where NULL would give the latest value.

Updates to the block and state tables happen atomically, within a single database transaction. This ensures that the block change and all the state changes remain consistent with the state of the validator network.

Examples

For example, a simple export of Intkey may look like, at block number 3:

id	intkey_name	intkey_value	start_block_num	end_block_num
1	count_a	1	1	NULL
2	count_b	1	1	2
3	count_b	10	2	NULL
4	count_c	15	3	NULL

Querying the latest values, i.e. the state values at the current chain head, from this table would be as follows:

```
SELECT intkey_name, intkey_value
FROM integer_key
WHERE end_block_num IS NULL;
```

The result would look like:

intkey_name	intkey_value
count_a	1
count_b	10
count_c	15

Querying the values at a particular block number would be as follows:

```
SELECT DISTINCT ON (intkey_name)
       intkey_name, intkey_value
FROM integer_key
WHERE :block_num >= start_block_num AND
  (:block_num <= end_block_num OR end_block_num = NULL)
ORDER BY intkey_name, end_block_num desc NULLS FIRST;
```

where `:block_num` is the block number of interest.

Using the above query, in combination with querying for the latest block number from the block table, a consumer of state data tables can ensure that they are receiving a consistent view of the validator state at a given block number. Simply querying a state data table for the latest, while it will be consistent for the query (thanks to using transactions while updating), information on when state has changed is lost, so records may appear and disappear depending on when a query is made. Note, the above query is effecient for small data sizes

Storage Solution Recommendations

It is highly recommended that the database support multi-version concurrency control (MVCC). This will allow the values from a `StateDeltaEvent` to be updated within a transaction, while still allowing other clients to read the database with the values for the current block.

It is also recommended that the database choice supports row compactions without locking the tables. This is useful in ecosystems where there are frequent forks and subsequent resolutions, which could result in frequent deletes.

PostgreSQL is a good recommendation because it provides both [MVCC]() and supports row compaction through [vacuuming]().

Open Issues

State Checkpointing

Once state checkpointing is implemented, state deltas before a certain point on the chain will be lost. This will require the client to use the prepopulate enhancement mentioned above, as requiring the validator to rebuild the delta store would defeat the purpose of state checkpointing.