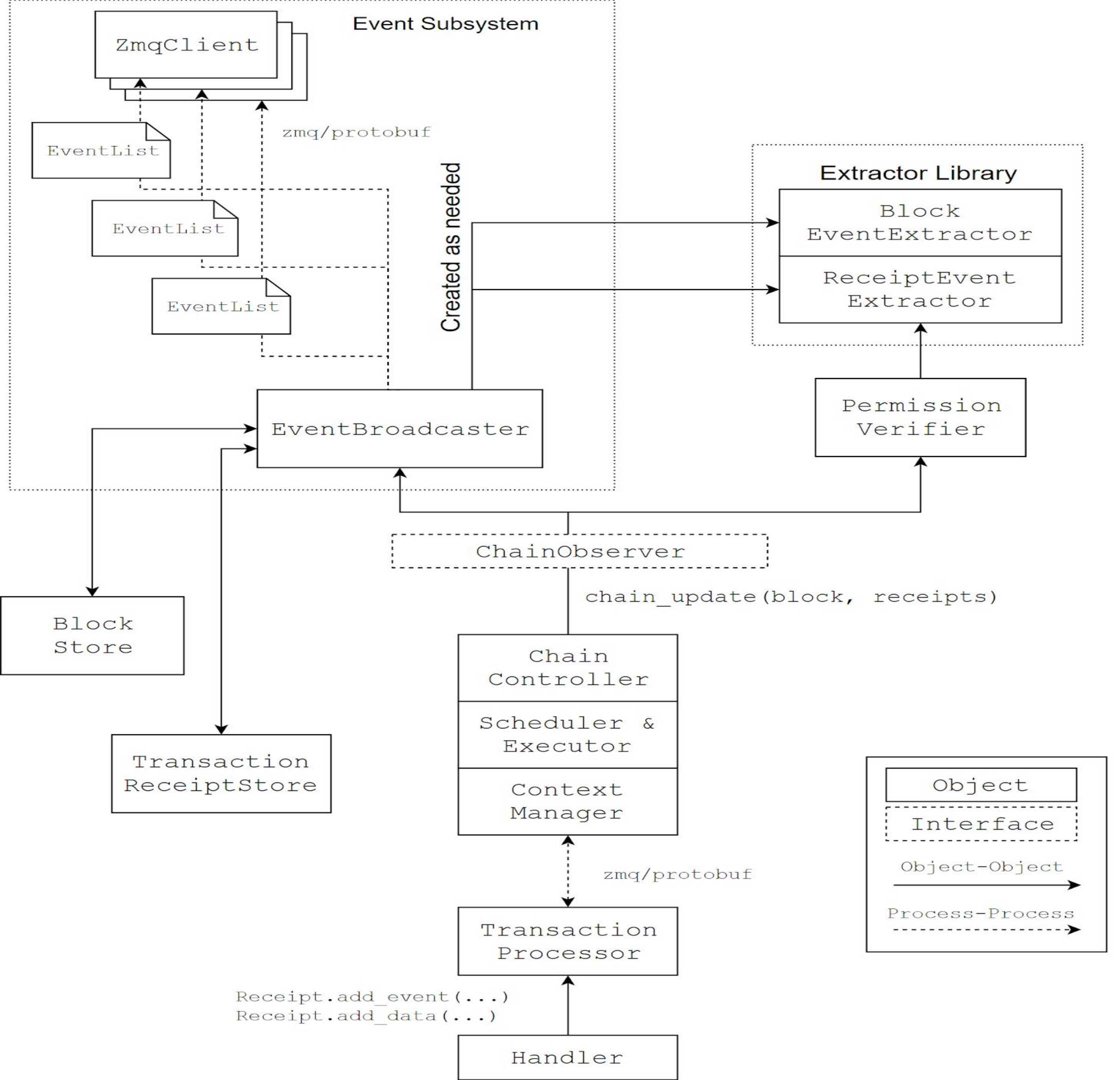
Events & Event Subscriptions

Events

- Hyperledger Sawtooth supports creating and broadcasting events.
- Subscribe to events that occur related to the blockchain, such as a new block committed or switching to a new fork
- Relay any information across the network, without storing that in state.
- Subscribe to application specific events


Event Subsystem
ZmqClient
zmq/protobuf
EventList
EventList
EventList
Created as needed
EventBroadcaster
Extractor Library
Block EventExtractor
ReceiptEvent Extractor
Permission Verifier
ChainObserver
chain_update(block, receipts)
Block Store
Transaction ReceiptStore
Chain Controller
Scheduler & Executor
Context Manager
zmq/protobuf
Transaction Processor
Receipt.add_event(...)
Receipt.add_data(...)
Handler
Object
Interface
Object-Object
Process-Process

Event structure

event_type (string)
attributes (array)
data (bytes)

```
message Event {
  string event_type = 1;

  message Attribute {
    string key = 1;
    string value = 2;
  }
  repeated Attribute attributes = 2;

  bytes  data = 3;
}
```

Event structure

- An event message consists of
  - event_type – name/identifier of the event
  - attribute – zero or more predefined data for the data_type
  - data – family specific opaque data defined for the event_type
- For event_type naming, the convention is to use a combination of family name and event name separated by “/”, to distinguish events of different transaction families.
- Sawtooth has block-commit and state-delta events, prebuilt

Sawtooth events

Sawtooth/block-commit

```
Event {
  event_type = "sawtooth/block-commit",
  attributes = [
    Attribute { key = "block_id", value = "abc...123" },
    Attribute { key = "block_num", value = "523" },
    Attribute { key = "state_root_hash", value = "def...456" },
    Attribute { key = "previous_block_id", value = "acf...146" },
  ],
}
```

Sawtooth events

Sawtooth/state-delta

```
Event {
  event_type = "sawtooth/state-delta",
  attributes = [Attribute { key = "address", value = "abc...def"
}],
  event_data = <bytes>
}
```

Creating a custom event

- Custom events can be created in TPs and listened from Client
- To create custom events in transaction processors, we use the ***addEvent*** API of the context object (ie the context object that we receive in the apply function).

context.**addEvent(eventName, list of attributes, any data)**

Example: For a text appending Transaction Processor

context.*addEvent(*

“<Helloworld/write>”,

[[“Text”, <current text appended>]],

“<byte encoded form of full text value”

)

Event Subscriptions

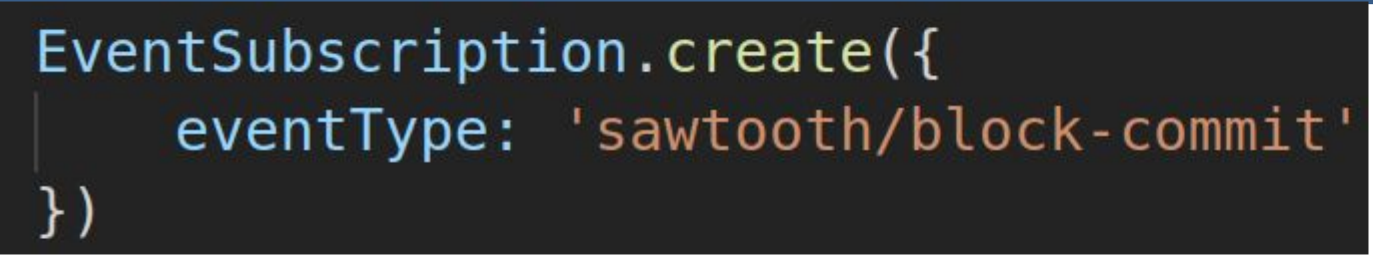
- The steps involved
  - Construct subscriptions for all the events that we want to listen to.
  - Send the list of subscriptions as a **ClientEventSubscribeRequest** protobuf message directly to validator
  - Validator will send back a **ClientEventSubscibeResponse** for the subscription.
  - If the **ClientEventSubscribeRequest** was correctly formed and the validator was able to register the subscriptions, **ClientEventSubscibeResponse** will have an "OK" response. Otherwise the response will be an error
  - Client will listen to events from validator. Client have a handler function that will process the event messages from the validator.

Subscriptions structure

event_type (string)
filters (EventFilter)

```
message EventSubscription {
  string event_type = 1;
  repeated EventFilter filters = 2;
}
```

Subscriptions structure



```
EventSubscription.create({
    eventType: 'sawtooth/block-commit'
})
```

```
message EventSubscription {
  string event_type = 1;
  repeated EventFilter filters = 2;
}
```

Event Filters

- A key/value pair meant for matching attributes of an event
- Can be used when we want to subscribe to only certain events of an event_type based on their attributes.
- A subscription can have a list of event filters of following type
    - *SIMPLE_ANY* – string match of filter key/value with any attribute value of an event
    - *SIMPLE_ALL* - string match of filter key/value with all attribute value of an event
    - *REGEX_ANY* - regex match of filter key/value with any attribute value of an event
    - *REGEX_ALL* – regex match of filter key/value with all attribute values of an event

Subscriptions structure

event_type (string)
filters (EventFilter)

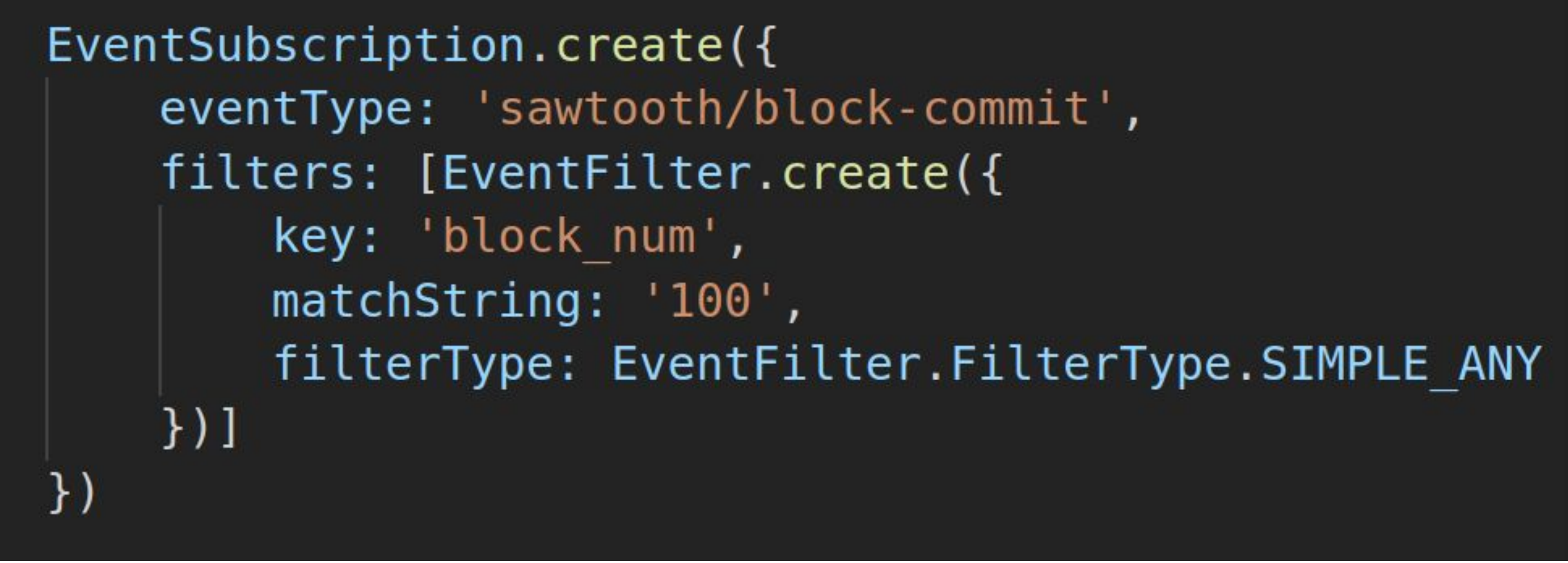
key (string)
match_string (string)
filter_type (FilterType)

```
message EventSubscription {
  string event_type = 1;
  repeated EventFilter filters = 2;
}

message EventFilter {
  string key = 1;
  string match_string = 2;

  enum FilterType {
      FILTER_TYPE_UNSET = 0;
      SIMPLE_ANY = 1;
      SIMPLE_ALL = 2;
      REGEX_ANY  = 3;
      REGEX_ALL  = 4;
    }
    FilterType filter_type = 3;
}
```

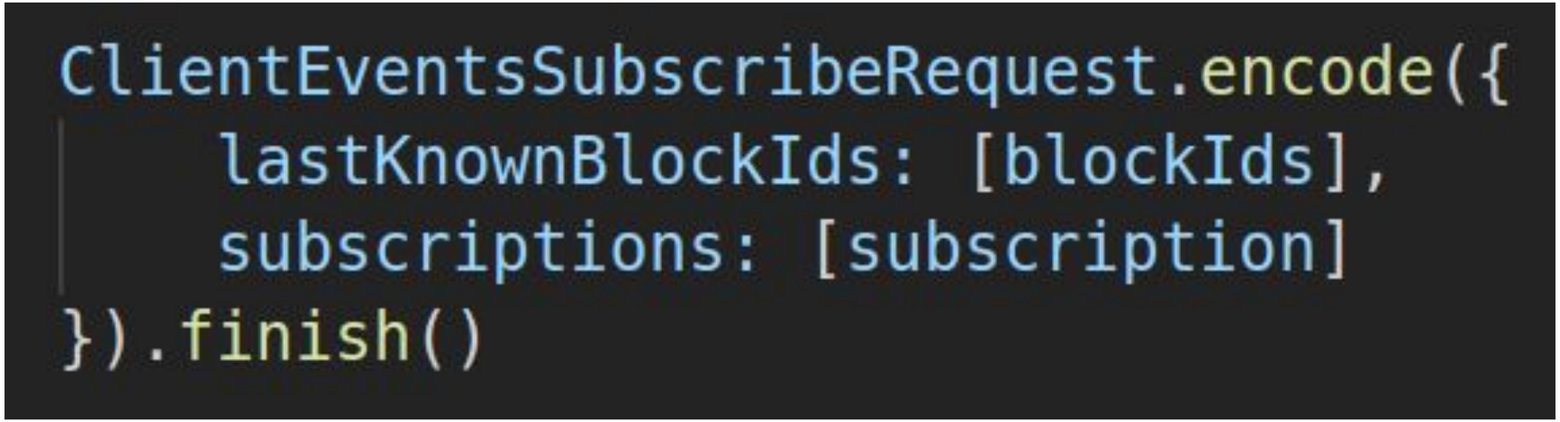
Create Subscription



```
EventSubscription.create({
    eventType: 'sawtooth/block-commit',
    filters: [EventFilter.create({
        key: 'block_num',
        matchString: '100',
        filterType: EventFilter.FilterType.SIMPLE_ANY
    })]
})
```

Create EventSubscriptionRequest

```
message ClientEventsSubscribeRequest {
    repeated EventSubscription subscriptions = 1;
    repeated string last_known_block_ids = 2;
}
```



```
ClientEventsSubscribeRequest.encode({
    lastKnownBlockIds: [blockIds],
    subscriptions: [subscription]
}).finish()
```

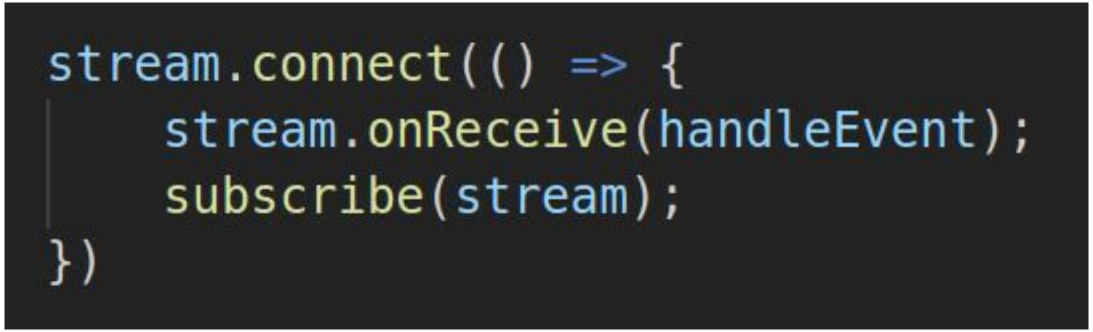
Stream connection

- So, we created a subscription request. Now we need to send it to the validator.
- We use the **Stream** class available at ‘***sawtooth-sdk/messaging/stream***’ of the javascript SDK, to make connection to the validator.

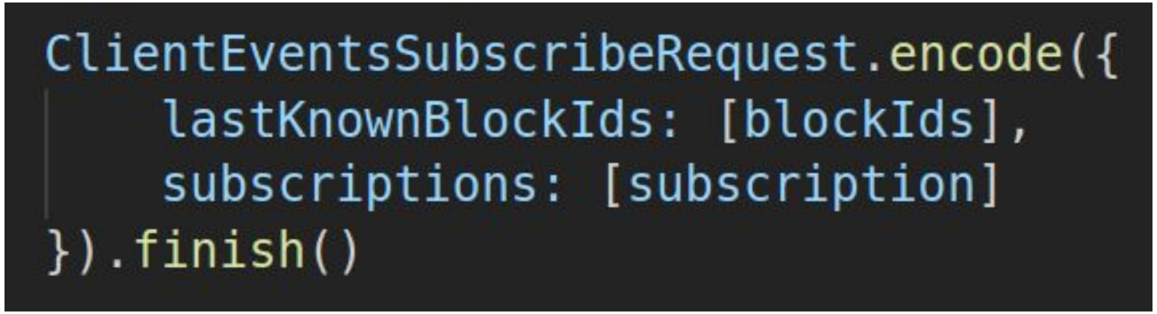
- ***const stream = new Stream("tcp://validator:4004)***
  Will create a new instance of **Stream** class with the address of the validator
- ***stream.connect()***

  Will create a new stream connection to the validator using the instance

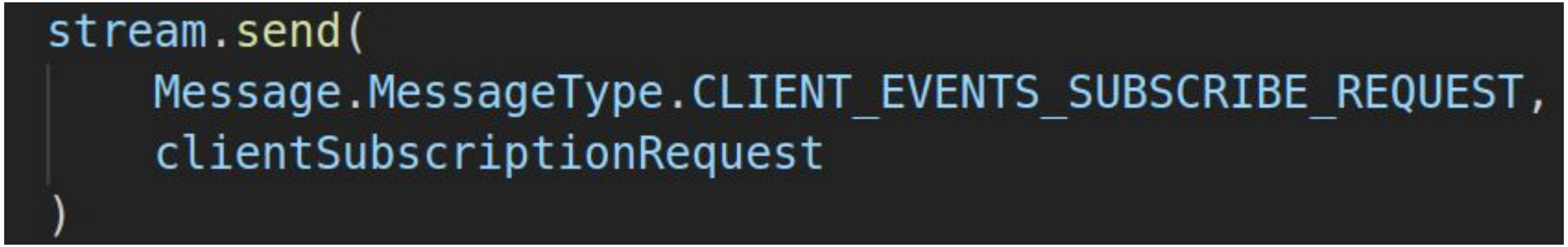
- stream.connect(callback) - The “callback” here, is a function where we will write what we should do once a connection with validator is made, for example, the subscription for events using this stream connection.



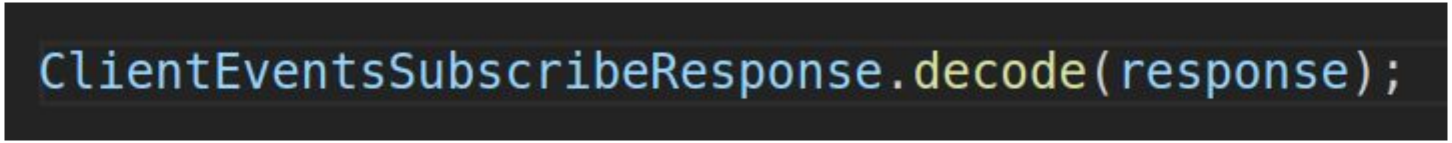
```
stream.connect(() => {
    stream.onReceive(handleEvent);
    subscribe(stream);
})
```



```
ClientEventsSubscribeRequest.encode({
    lastKnownBlockIds: [blockIds],
    subscriptions: [subscription]
}).finish()
```



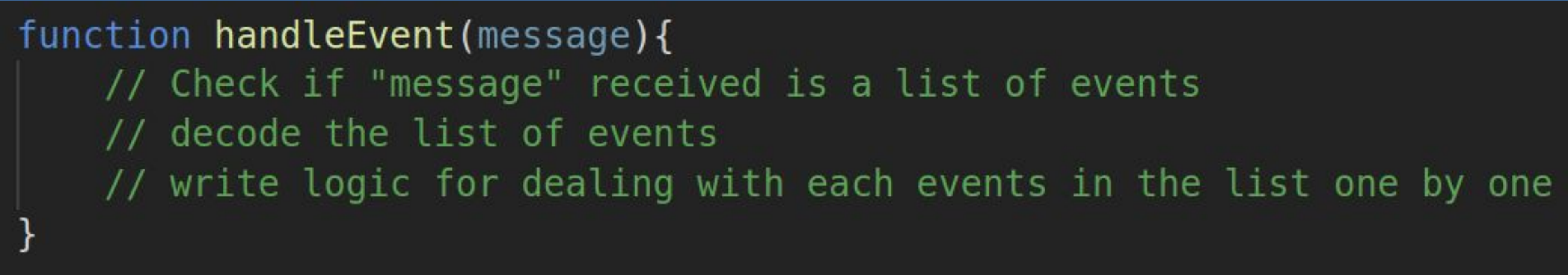
```
stream.send(
    Message.MessageType.CLIENT_EVENTS_SUBSCRIBE_REQUEST,
    clientSubscriptionRequest
)
```



```
ClientEventsSubscribeResponse.decode(response);
```

```
message ClientEventsSubscribeResponse {
    enum Status {
         OK = 0;
         INVALID_FILTER = 1;
         UNKNOWN_BLOCK = 2;
    }
    Status status = 1;
    // Additional information about the response status
    string response_message = 2;
}
```

Event Handler



```
function handleEvent(message){
    // Check if "message" received is a list of events
    // decode the list of events
    // write logic for dealing with each events in the list one by one
}
```

- The “message” that we get in our handler function is an **EventList** protobuf message.
- An eventlist is a list of **Event** message
- Remember how an **Event** looks like (from our initial slides), with event_type, attributes and data.

Event subscription using REST API

- Using a standard web socket connection with the REST API from the client.
- Custom events are not supported .
- Event filtering is not possible.
- Event catch up functionality is not available.
- Event catchup functionality - We can get events from previous blocks using the “lastKnownBlockIds” property in a **ClientEventsSubscribeRequest**

K B A

Transaction Receipts

Transaction receipts

- Provide information about transaction execution that doesn’t need to be stored in state
- Transaction validity, state changes, events generated and other Family specific info that need not be stored in state.
- This is useful to retrieve past transaction execution informations, like events, state changes without executing transaction again
- The ‘TransactionReceiptStore’ stores the transaction receipts in an off-chain store. (This is a separate DB)
- Clients can request transaction receipts for a transactionId from the REST API

```
message TransactionReceipt {
  repeated StateChange state_changes = 1;

  repeated Event events = 2;

  repeated bytes data = 3;

  string transaction_id = 4;
}
```

Adding Transaction Receipt data

- You can store application specific data regarding a transaction in the transaction receipt (ie in the ***data*** field)
- We can use the addReceiptData API of TP context object to add data to a transaction receipt

THANK YOU

Questions ?