Reading Metering Records from the Blockchain

Overview

Metering records are sent from data consumers to data producers periodically, indicating a period of time in which valid data is received by the data consumer from the producer. The metering record is contained within a secured metering notification. A metering notification is like a signed (by the data consumer) paper check that the data producer can cash at any time by writing it to the blockchain.

Metering records are written to the blockchain to publicly record that a data producer has sent the agreed to data to a data consumer. While the records are written by the data producer, the data consumer provides metadata (verified by the blockchain smart contract) proving that it has agreed to both the encompassing agreement in which the metering record is being written, as well as the metering record itself.

Agreements between producers and consumers that include data metering, will indicate a number of metering tokens per unit time that are to be granted while the agreement is in effect. At the end of the agreement, the producer writes the last metering record it received into the blockchain. The tokens granted to the producer have no inherent value. They are simply an indication that the data consumer was receiving valid, agreed to data over some period of time from the producer. The tokens could be converted to something of value outside the Horizon platform.

Here is what a metering notification message looks like conceptually, rendered in JSON for readability. These message are never visible on the network. It is included here to aid in understanding the metering concept. The consumer_mp_signature is the signature of the hash of the amount, current_time and agreement_id. This is the signature that pre-authorizes the producer to write the metering record to the blockchain.

```
{
 "amount": 121,                                  // number of tokens
 "start_time": 1491487140,                       // number of seconds since 1970
 "current_time": 1491487342,                     // number of seconds since 1970
 "missed_time": 45,                              // number of seconds of missing data
 "agreement_id": 12345678901234567890,
 "consumer_mp_signature": "lIUBfd764lfsie=",     // consumer's signature of the metering
msg
 "agreement_hash": 111222333444,                 // the hash of the merged policy agreement
 "consumer_hash_signature": "aweywqnfwoiyu4834=", // consumer's signature of the hash
 "consumer_address": "0x456789abcdef",           // the consumer's ethereum address
 "producer_hash_signature": "3klvnejHEYUE="      // producer's signature of the hash
}
```

Reading Metering Records

When the data producer writes a metering record to the blockchain, an event is added to the blockchain block that contains the execution of the write transaction. Anyone can read these blockchain events, and anyone can verify the content of the records. Extracting the events requires two things: the use of the ethereum RPC API to read events, and knowledge of the event format that Horizon is using for the body of the event.

Horizon Smart Contract Addresses

Establishing a new event filter requires the caller to know the smart contract address of the smart contract which issued the event. For this, some Horizon platform blockchain bootstrap information is needed. The Horizon platform currently has 3 smart contracts; Directory, Agreements, and Metering. The Directory contract holds the addresses of the other 2 smart contracts.

1. To get the address of the directory contract, query the blue horizon ethereum blockchain bootstrap information stored in SoftLayer object store:
    a. https://dal05.objectstorage.softlayer.net/v1/AUTH_773b8ed6-b3c8-4683-9d7a-dbe2ee11095e/bluehorizon/directory.address
2. With the directory address, use the ethereum rpc eth_call method to invoke the “get_entry_by_version” method on the Directory smart contract to get the Metering smart contract address:
    a. You do **not** need to specify parameters: from, gas, gasPrice, value.
    b. Set the 2nd parameter (QUANTITY) to "latest"
    c. Set data in the object of the 1st parameter to the call signature and parameters of get_entry_by_version using the Ethereum Contract ABI format.
        i. Set entry_name to “metering” to obtain the address of the metering smart contract.
        ii. Set version to 0 (zero)

iii. The metering contract address is needed when creating a new ethereum event filter in the next section.

Creating an Event Filter

You will use several methods from the ethereum RPC API to create and then use an event filter to extract the metering records from the ethereum blockchain.

A simplified sequence for reading metering events looks like this:

1. Use eth_newFilter to create a new event filter, passing it:
    a. address: the metering smart contract from above
    b. fromBlock: query the current block number from any edge device using curl -sS http://localhost/status | jq .geth, subtract enough blocks to get to before your agreement ended (estimate 20 seconds per block), and convert the block number to hex.
    c. toBlock: if the agreement ended recently, set this to `latest`
    d. topics: set the last of the 4 elements of this array to the agreement id you are interested in to reduce the output, e.g. [null,null,null,"0x68e5d9e637c20569137f7c4a53a6969c3feac572cc202af50e128912b0d18740"]
    e. The result field in the json returned by this rpc call will be the filter object address, which you pass into the next API call.
    f. Example rpc call:

```
export METERING_ADDRESS=0xe9f3a8554b24cbfa859d23da89ab752e89353e45
export
AGR_ID=0x68e5d9e637c20569137f7c4a53a6969c3feac572cc202af50e128912b0d18740
export FROM_BLOCK=0xC6B14
curl -X POST --data
"{\"jsonrpc\":\"2.0\",\"method\":\"eth_newFilter\",\"params\":[{\"fromBlock\"
:\"$FROM_BLOCK\",\"toBlock\":\"latest\",\"address\":\"$METERING_ADDRESS\",\"t
opics\":[null,null,null,\"$AGR_ID\"]}],\"id\":73}" localhost:8545
```

    g.
2. Invoke eth_getFilterLogs to retrieve events found by the above filter:
    a. params: a 1 element array containing the filter address from above
    b. Example rpc call:

```
export FILTER_ADDR=0x3aa7d4702d12bd6be3e9996627f6c5d2
curl -X POST --data
"{\"jsonrpc\":\"2.0\",\"method\":\"eth_getFilterLogs\",\"params\":[\"$FILTER_
ADDR\"],\"id\":74}" localhost:8545
```

3. Process each event (see below).
4. Optionally delete the filter with the eth_uninstallFilter() API (https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_uninstallfilter). Ethereum will clean this up in time, but cleaning them up proactively is a good idea.
5. Loop back to (1) to process the next batch of blocks or agreement ID.

Blockchains continuously get longer and longer, therefore it is the blockchain reader's responsibility to keep up with new blocks as they arrive. The tricky part of this is that forks in the chain can lead a blockchain reader to a false conclusion. Therefore, it is wise to never read from the top 3 to 5 blocks on the chain. As a result, an event processing loop needs to know what the current block is, but it should set the "toBlock" parameter in the eth_getNewFilter API to current block minus 3 (or 5 for even more safety). It might be tempting to try to use the eth_getFilterChanges API. The problem with using that API is that it always reads from the newest block in the chain, so the results returned are subject to be invalidated in the near future if a fork is resolved differently that what was just read from the event filter.

Processing an Event

The response to the eth_getFilterLogs() API returns an array of events. There are primarily 3 fields of interest within each returned event; "address", "data", and "topics". The "address" field will always be the same as the metering smart contract address used to create the filter. If it's different then ethereum has a bug.

The "topics" field contains contextual information related to a smart contract metering event. The Solidity code for the metering contract is in open source (https://github.com/open-horizon/go-solidity/blob/master/contracts/metering.sol). Within the solidity source you can see the event definitions (there are currently 4). Of primary interest is the "CreatedMeterDetail" event, the definition of it is expanded upon in this document.
The first 4 fields of the event are placed into the "topic" array of the events returned from eth_getFilterLogs. That is, for each event returned, the "topics" array contains (in string encoding):

```
[0] - 256 bit number set to 1, indicating that it is the created meter detail event
[1] - ethereum account address of the data producer (160 bit number)
[2] - ethereum account address of the data consumer (160 bit number)
[3] - 256 bit number which is the agreement id of the agreement between these 2 parties
```

It is important to remember that anyone can write records into the metering smart contract, and therefore not all records are necessarily valid. It is important for a data consumer to know which ethereum accounts it's Agreement Bot(s) are using so that the process which is reading records can ignore events related to ethereum accounts that it is not interested in.

Decoding the "data" section of the event is more complicated. The data section is a string encoded bit string in the format defined by ethereum called ABI (https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI). Here is how to breakdown the data field. First, the field is always a multiple of 32byte fields in length. Second, there are 2 sections within it, the header section followed by the variable length values. For the CreatedMeterDetail event, the data field looks like this:

```
000-031 - token count
```

```
032-063 - time record is created on the agbot - number of seconds since 1970
064-095 - hash of the meter record
096-127 - offset into the data section of the consumer's signature of the meter record hash
128-159 - hash of the agreement (merged policy document)
160-191 - offset into the data section of the producer's signature of the agreement
192-223 - offset into the data section of the consumer's signature of the agreement
---------------- variable length values start here ------------------
224-351 - consumer's signature of the meter record hash (length followed by signature)
352-479 - producer's signature of the agreement (length followed by signature)
480-607 - consumer's signature of the agreement (length followed by signature)
```

Here is an example of an event returned from eth_getFilterLogs() on a private test system. Note that the consumer and producer account addresses are the the same, this is due to the configuration of the test environment, it is not expected in a real world deployment. More importantly note the 4 topics fields and the long stringified bit string in the data field, which is formatted in this document into 32 byte segments for readability.


```
{"jsonrpc":"2.0","id":"1","result":[
{"address":"0x3a7389c5aedc337fe4cff73cb7de5423a1bdccd4",
"Topics":["0x0000000000000000000000000000000000000000000000000000000000000001",
"0x000000000000000000000000b45c43b469a8266e818bbdd6311919abda0c22c6",
"0x000000000000000000000000b45c43b469a8266e818bbdd6311919abda0c22c6",
"0x9ff57c36d1b60b62db92cec8fe210abe2322c2f2aab17438abdf9de286260bf9"],
"Data":"0x0000000000000000000000000000000000000000000000000000000000000009
0000000000000000000000000000000000000000000000000000000005907314f
3acc8dd86070f87512c6c983d6ca9bb1d08eda774c491d9ffd20be06248c31a4
00000000000000000000000000000000000000000000000000000000000000e0
033de044bad817b6acbc5f204c4d3f1711a1e275ca7903cdb78714f2be86492e
0000000000000000000000000000000000000000000000000000000000000160
00000000000000000000000000000000000000000000000000000000000001e0
0000000000000000000000000000000000000000000000000000000000000041
f07ad9c1cd0a04c99fc632913e3a0557a4ce3976cf2edf9f10d688bc6cf6b29d
038ee7c5a5049123f3a3f14a68c2d91175e041b42fa7d1d20cd01fe6d7a7f147
1b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000041
c2e4f19bea6139872b4a364c610a80f09502ca93e39183e15869ba6e427577d7
73c25aef4f563b1387c7248a31a3604c7ea4810b163c8ca1b0db8e707759e2c5
1b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000041
c2e4f19bea6139872b4a364c610a80f09502ca93e39183e15869ba6e427577d7
73c25aef4f563b1387c7248a31a3604c7ea4810b163c8ca1b0db8e707759e2c5
1b0000000000000000000000000000000000000000000000000000000000000000",
"blockNumber":"0x7f","transactionIndex":"0x1","transactionHash":"0xe2de16168c758d6d9056a7d10
7ebe96f5ad7d49494f47fbe14baf20bd2d2e3c6","blockHash":"0x3e77673fae9431b5920649684c01db348b37
5385c6027c8a9c2060c18981b2b3","logIndex":"0x2","removed":false
}]}
```


The important piece of data to correlate with is the meter record hash because that is the thing that is signed by the consumer. The AgreementBot, acting on the data consumer's behalf will have a record of the last metering notification that was sent to the data producer. As long as there is a match between the AgreementBot's hash and meter record hash in the event, then the data consumer should consider the event a valid event. There is more correlation that can also be done, but is not elaborated within this document.

The Metering smart contract will ONLY write the meter record onto the blockchain IF all the following are true:

a) The submitter of the transaction is the data producer
b) The submitter of the transaction signed the agreement hash
c) The submitter of the transaction provided the ethereum account of the data consumer and the consumer signed both the meter record hash and the agreement hash.

It is possible for anyone with an ethereum client to verify the signatures in the metering events. The hashes are intended to obscure the actual data that is being signed as it is private to the 2 parties in the agreement.

It should be noted that the token count field and the timestamp field are present in the record for testing/simplicity but should not be relied on. A rogue producer could have written in false values, but they could not have changed the hash without invalidating the consumer's signature of it. As long as the consumer verifies metering records by meter record hash, then everything is secure.

Correlation with an Agreement Bot

Agreement Bots (agbots) act on behalf of a data consumer to make agreements, monitor data receipt at the data ingest and to provide metering notifications (a meter record is contained within a metering notification) when enabled. Agbots record agreements and metering records within a local database. A data consumer needs to access these records in order to verify agreements and metering records on the blockchain. The agbot has a REST API (similar to the device side API) that enables a data consumer to extract information about current agreements and agreements that have terminated. The API is only available to REST clients running on (and therefore already authorized to access) the agbot's host. The APIs are as follows:

1) GET /agreement - returns an array of active agreements and an array of archived agreements
2) GET /agreement/<agreement-id> - returns a single agreement record by agreement id
3) DELETE /agreement/<agreement-id> - deletes an archived agreement from the agbots local database

It is assumed that these APIs will be used by any agent that is reading blockchain records to process metering records and would like to verify the information with it's own agbots.

Archived agreements in an agreement bot contain the last 2 metering notifications (in the field called MeteringNotificationMsgs) that were sent to the producer. The producer will have written 1 of those 2 notifications to the blockchain when the agreement terminated. The agreement may have been terminated by the device while the agbot was sending a new metering notification, and therefore the newest notification sent by the agbot might not be written to the blockchain. When converting tokens to some external value, it is important to use the token counts in the agbot agreement records. The token quantity written to the blockchain cannot be trusted, only

the hash of the metering record can be trusted. So, as long as the agbot's meter record hash matches the hash in the blockchain, then the agbot's token count will be correct.

Summary

A summary of the steps for a data consumer to read blockchain metering records is:

1) Get the Horizon directory contract address.
2) Get the Horizon metering contract address.
3) Establish an event listener that reads blocks from the blockchain and processes them as appropriate as described above.
4) Establish an agbot agreement monitor that reads agreement records from the agbot and makes this data available to the event processor so that it can cross correlate/validate blockchain events. This monitor deletes archived agbot agreements that have been processed.