Storecoin Platform Smart Contract

Introduction

This specification describes the smart contract between data buyers, app developers, Storecoin dWorkers, and optionally app end users on how the revenue is derived from the decentralized app data and shared among the latter 3 parties. The smart contract approach ensures that Storecoin Platform remains independent of the base settlement layer, which is responsible for running the BlockFin consensus algorithm, processing the payment transactions, and securing the Storecoin blockchain.

The smart contract in this context shouldn't be confused with the similar concept in Ethereum and other blockchains. While it is designed as a general purpose tool, it is scoped to describing the revenue generated from the demand for the app data from data buyers and sharing the revenue between the 3 parties — app developers, Storecoin dWorkers, and optionally app end users — discussed above. The tokenized apps running on Storecoin Platform are general purpose webapps with a lightweight interface defined to connect them to the platform. So, no smart contract infrastructure is required to execute tokenized apps like the dApps in Ethereum.

Definitions

Tokenized apps — are web and mobile apps, whose decentralized, discoverable, structured data have intrinsic values and hence the data can be tokenized and sold to data buyers.
Data buyers — are entities who are interested in the structured data produced by the tokenized apps. The buyers may pay for the data in tokenized app's *datacoins* or base STORE token, depending on how the contract is arrived at.
Datacoins — are app tokens that represent the decentralized data produced by tokenized apps. Datacoins derive their value from the app data they are backing.
App developers — deploy their tokenized apps on Storecoin Platform. App developers may get the compute resources to run their apps from dWorkers (discussed next), if dWorkers find the app data valuable. In this scenario, there is no upfront cost to app developers to run their apps on Storecoin Platform. Instead, dWorkers agree to be compensated in the tokenized app's datacoins.
dWorkers — are the nodes that run tokenized apps. They also process payment transactions and secure the Storecoin blockchain in the base settlement layer. They are also called as miners.
App users — are the users of the tokenized apps. They produce the data, which are represented by datacoins. The users don't necessarily have to be humans; they can be robots, IoT devices, and other

kinds of data producers. Depending on the type of tokenized apps, the app users may be compensated for the data they produce in the app.

Of the 4 types of entities that participate in Storecoin Platform — data buyers, app developers, Storecoin dWorkers, and app end users — the first type, data buyers, pay to get the access to the app data. They are the *customers* of the app data. The latter 3 types are eligible to share the revenue generated by the tokenized apps.

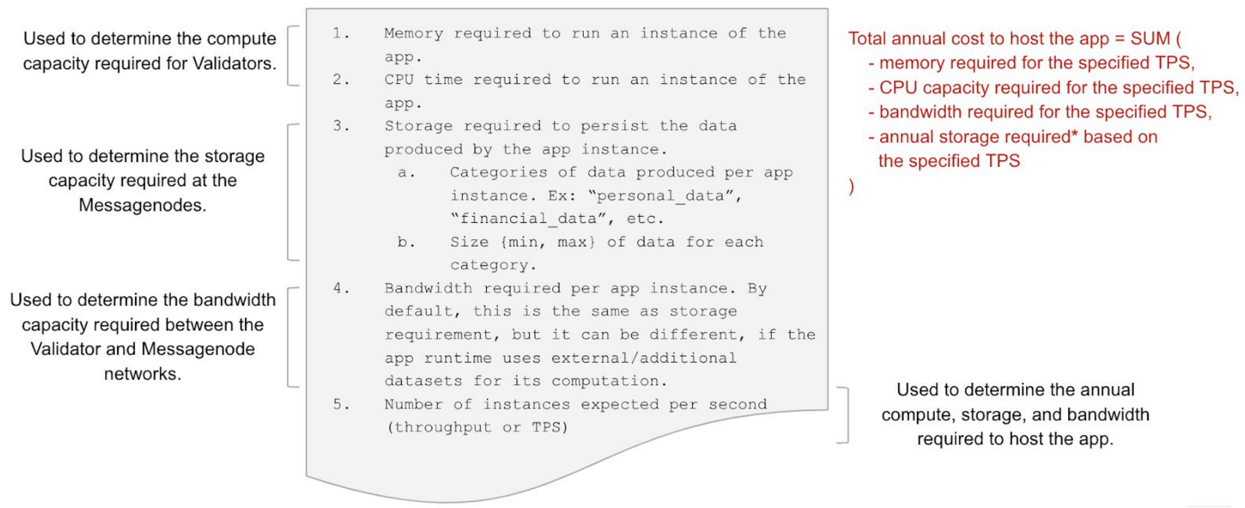
Why smart contract?

At a high level, tokenized apps are approved as follows on Storecoin Platform.

1. The app developers submit a proposal to Storecoin dWorkers describing their apps. The proposal includes runtime requirements for the app (memory and CPU usage and storage and bandwidth requirements), anticipated throughput desired in number of app instances running concurrently per second, the structure of their data, and tiers of access or classes of the app data.
2. Storecoin dWorkers use the app proposal to determine the cost of hosting the app. The determination includes the annual cost of the runtime, storage, and the bandwidth required to support the app on Storecoin Platform.
3. Storecoin dWorkers also use the app proposal to determine the potential revenue from the decentralized app data. They will use a combination of market and income-based analysis to value the app data.
4. If the potential revenue is attractive, meaning it is more than the cost computed in step 1, plus a compute premium as profit, dWorkers approve the app on the platform. An *approval* means that the app developers will get free compute resources to host their apps in return for revenue sharing with dWorkers in app's *datacoin* — a custom app token that represents and backs the data in the app.
5. If dWorkers are unable to determine the value of the app data or if the data is not profitable to justify the cost of hosting the app, dWorkers reject the proposal. If app developers are motivated to deploy their apps on Storecoin Platform for better data discoverability and hence better revenue possibilities, they can *rent* compute resources by paying in base STORE to dWorkers.

Fig. 1 shows a sample of the app proposal submitted by app developers. dWorkers use the app proposal to carry out steps 1 and 2 discussed above.

Storecoin dWorkers determine the cost of hosting (runtime, storage, and bandwidth) the tokenized apps from the app proposals submitted by app developers.

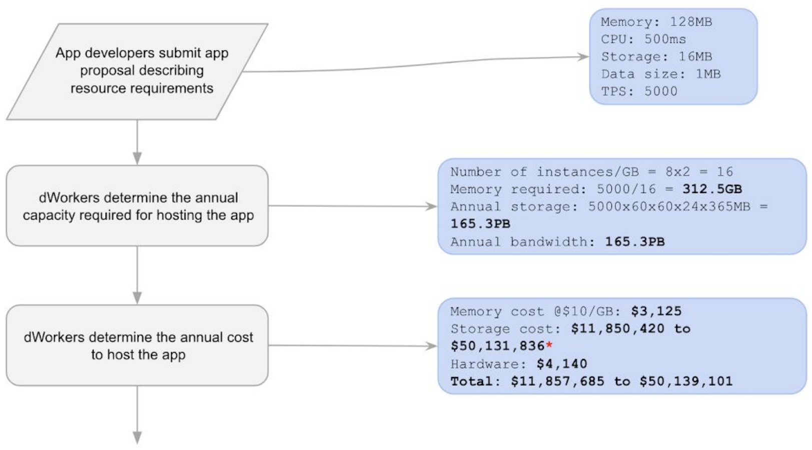

Used to determine the compute capacity required for Validators.
Used to determine the storage capacity required at the Messagenodes.
Used to determine the bandwidth capacity required between the Validator and Messagenode networks.
Used to determine the annual compute, storage, and bandwidth required to host the app.

```
1.   Memory required to run an instance of the
     app.
2.   CPU time required to run an instance of the
     app.
3.   Storage required to persist the data
     produced by the app instance.
  a.    Categories of data produced per app
        instance. Ex: "personal_data",
        "financial_data", etc.
  b.    Size {min, max} of data for each
        category.
4.   Bandwidth required per app instance. By
     default, this is the same as storage
     requirement, but it can be different, if the
     app runtime uses external/additional
     datasets for its computation.
5.   Number of instances expected per second
     (throughput or TPS)
```

Total annual cost to host the app = SUM (
- memory required for the specified TPS,
- CPU capacity required for the specified TPS,
- bandwidth required for the specified TPS,
- annual storage required* based on the specified TPS

)

Fig. 1 — Proposal submitted to dWorkers by app developers on resource requirements


App developers submit app proposal describing resource requirements
dWorkers determine the annual capacity required for hosting the app
dWorkers determine the annual cost to host the app

```
Memory: 128MB
CPU: 500ms
Storage: 16MB
Data size: 1MB
TPS: 5000
```

```
Number of instances/GB = 8x2 = 16
Memory required: 5000/16 = 312.5GB
Annual storage: 5000x60x60x24x365MB =
165.3PB
Annual bandwidth: 165.3PB
```

```
Memory cost @$10/GB: $3,125
Storage cost: $11,850,420 to
$50,131,836*
Hardware: $4,140
Total: $11,857,685 to $50,139,101
```

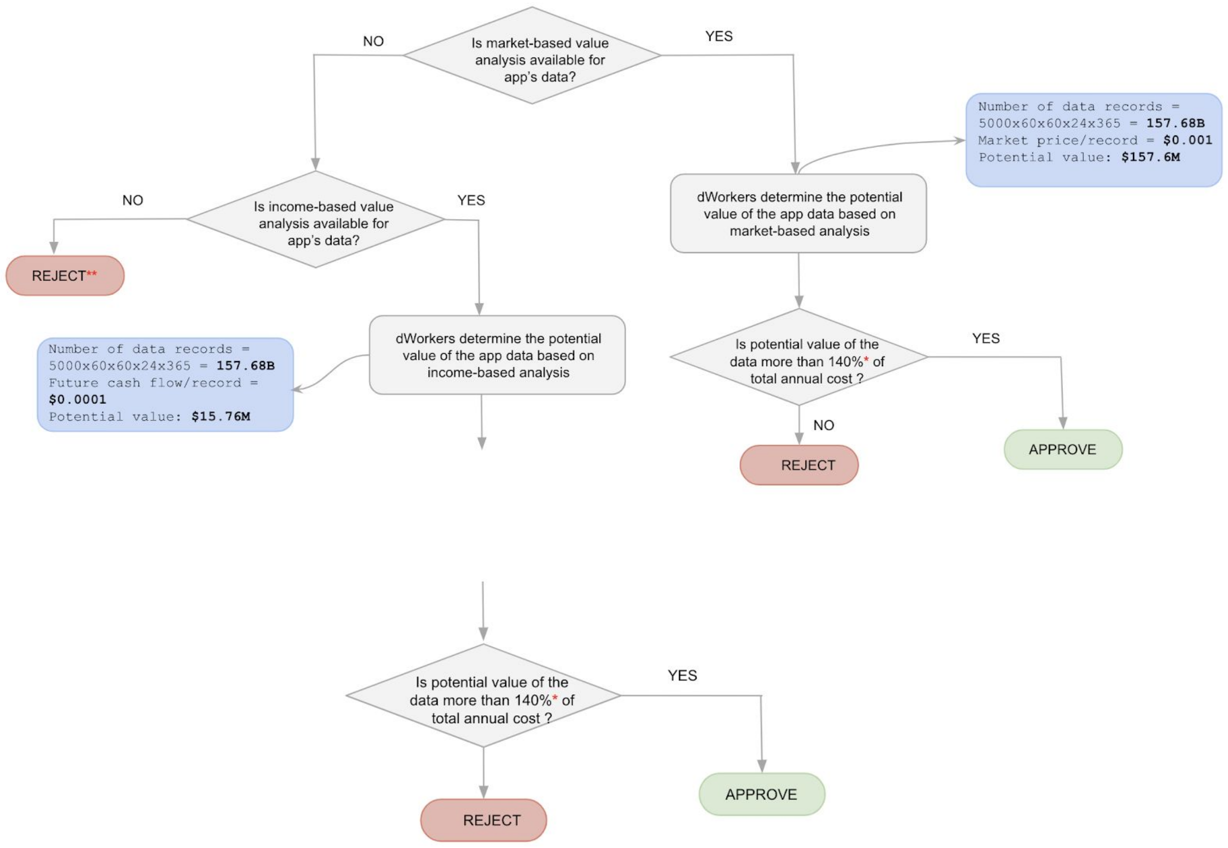

Is market-based value analysis available for app's data?
NO
YES
Is income-based value analysis available for app's data?
NO
YES
REJECT**
dWorkers determine the potential value of the app data based on market-based analysis
Number of data records = 5000x60x60x24x365 = 157.68B
Market price/record = $0.001
Potential value: $157.6M
dWorkers determine the potential value of the app data based on income-based analysis
Number of data records = 5000x60x60x24x365 = 157.68B
Future cash flow/record = $0.0001
Potential value: $15.76M
Is potential value of the data more than 140%* of total annual cost ?
YES
NO
REJECT
APPROVE
Is potential value of the data more than 140%* of total annual cost ?
YES
REJECT
APPROVE

Fig. 2 — App data valuation process

Fig. 2 illustrates the process that dWorkers use to value the app data, which leads to the eventual approval or rejection of the app proposal.

Based on the results of steps 4 and 5, app developers may end up getting free compute resources provided by dWorkers in exchange for payments in datacoins (the revenue sharing model) or end up renting the compute resources from dWorkers by paying them in base STORE token (the rent model).

The decision to provide free compute resources in exchange for revenue sharing in datacoins or renting the resources for a payment in STORE needs to be codified for automatic execution in the platform. This is because the approval for each app will likely be different due to their resource requirements, data valuation, and so on. The following steps are required in order to ensure that the resource usage is metered correctly and dWorkers are paid for securing the tokenized apps as promised by the app developers.

1. **Meter resource usage** — The apps may receive unlimited compute resources or resources of certain capacity for certain duration. In any case, accurate metering is necessary so that

dWorkers can continuously monitor the resource usage and app developers are certain that the usage reported is accurate. The latter is especially important when the compute resources are rented by the app developers. The result of metering the resource usage is also used for capacity upgrade, if needed, or to optimize the resource usage by dWorkers.
2. **Meter revenue in datacoins** — When the app data is purchased by data buyers, the buyers pay for the data in datacoins. Each transaction that requests and accesses the data either pays for the data directly or uses a previously granted authorization token to access the data. A payment or subscription is required to grant the authorization token. In either case, the revenue realized by the app data must be metered so that the dWorkers get their share of the revenue.
3. **Ensure *trust* in both the metering steps** — It is possible for malicious dWorkers to overstate the resource usage or malicious app developers to understate the revenue realized by their app data. So, the computations must be *trusted* by both the parties. This can be achieved only if the computations are automatic and happen in the Storecoin Platform layer.

The revenue sharing and the rent models require slightly different smart contracts because of the way dWorkers are compensated. In the revenue sharing model, dWorkers are compensated on an ongoing basis based on the revenue produced by the tokenized app. In the rent model, app developers may pay for the required resources in advance for better price (say, rent the compute resources for a year) or pay on an ongoing basis based on the resources used. But in the rent model, dWorkers are compensated in base STORE token.

For simplicity, we describe the rent model first as the process to comparable to traditional cloud service providers, such as AWS. The tokenized apps may transition from one to the other model depending on the business needs and how apps evolve on the platform. For example, a tokenized app may start with the rent model and prove its worth to dWorkers, who initially would have rejected the app proposal for lack of data valuing methodologies for specifics of the app data. Once dWorkers can determine the value of the data, the app may switch to the revenue sharing model. The reverse may also happen. A tokenized app may not generate the required revenue to make hosting the app profitable to dWorkers. In this case, the app may switch to the rent model from the initial revenue sharing model.

Smart contract for the rent model

Listing 1 illustrates the smart contract for the rent model. In this model:

- the app data may have a value, but dWorkers decline payments in the app's datacoins. App developers pay for the compute resources in base STORE token.
- app developers are free to decide how they generate revenue. They may still use datacoins and require data buyers to pay for the data in datacoins, require payments for the data in

base STORE, or a fiat currency. Storecoin Platform is still used to enforce all the rules for data access.

- dWorkers meter the compute resources to ensure that the usage doesn't exceed the promised capacity. They don't care about the revenue generated by the apps.

For the sake of simplicity we assume that the app, which decides to rent resources doesn't have its own datacoin and data buyers pay for the data in STORE or fiat. It is easy to support the case for datacoins with the smart contract designed for the revenue sharing model, discussed later in this specification.

```
{
  # Body of the contract.
  body: {
    # Datacoin name or symbol.
    name: <name of the datacoin>,

    # Version number of the contract. The contract may go through multiple revisions
    # in its lifetime.
    version: <version number>,    # Ex: "1.0.1".

    # Developer's wallet address is recorded in the contract. The payments from
    # data buyers is deposited to this address and payments to dWorkers is
    # withdrawn from this address.
    developer_address: <App developer's wallet address>,

    dWorker_address: <A wallet address representing the dWorker group>,

    # Rent for the compute resources. The resources can be used on-demand or reserved
    # for a longer duration such as a year. The prices are in STORE. The rent
    # information is used to determine how much the app developers must pay
    # the dWorkers for using the compute resources.

    rent: {
      # On-demand pricing in STORE. The prices are captured for the basic units of
      # respective compute resources. If this method is chosen, app developers
      # are billed for the resources they actually used.
      on_demand: {
        memory_per_gb: <Price in STORE for 1GB for memory>,
        cpu_cycle: <Price in STORE for 1 CPU cycle = CPU usage for 1 second
                    of app execution>,
        bandwidth_per_gb: <Price in STORE for 1GB for bandwidth>,
        storage_per_gb: <Price in STORE for 1GB of permanent storage>,
        throughput: <Allowed concurrency measured in transactions/second>,
        billing_period: <DAILY|WEEKLY|MONTHLY>
      },

      # Reserved price in STORE. The data is same as above, but the price
      # reflects a long term commitment. If app developers choose this approach,
      # the payment is required in advance for the chosen reserve period.
      reserved_1_year: {
        memory_per_gb: <Price in STORE for 1GB for memory>,
        cpu_cycle: <Price in STORE for 1 CPU cycle = CPU usage for 1 second
                    of app execution>,
        bandwidth_per_gb: <Price in STORE for 1GB for bandwidth>,
        storage_per_gb: <Price in STORE for 1GB of permanent storage>,
        throughput: <Allowed concurrency measured in transactions/second>,

        # The specific resources granted for this app are listed here.
        compute_resources: {
          memory_in_gb: <Runtime memory in GB>, # Ex: 256.00
          # A "cycle" is CPU usage for 1 second of app instance execution.
          cpu_cycles: <Annual CPU cycles reserved>,
          bandwidth_in_gb: <Annual bandwidth available in GB>,
          storage_in_gb: <Annual storage guaranteed in GB>,
          throughput: <Number of concurrent app instances purchased in
                       transactions/second>
        }
      },
      . . .
    },

    # Data access rules. These rules are enforced to ensure that the buyers pay
    # for the data based on their contract agreement with app developers and/or
    # dWorkers. Storecoin Platform spec defines different data access "tiers",
    # so buyers may choose appropriate tier based on their needs. The tiers
    # themselves are app specific, but the price specified for the data for each tier
    # will be enforced by the platform.

    data_price: {
      # The price of the data for various tiers. Data buyers will be requesting the
      # data for specific tiers, so the payment can be enforced before access is
      # granted. The tier names here are up to the apps.
      tier_1: {
          "fields": {
             # Data fields (and their structure) returned in this tier.
             "field1": {
                "type": "string",
                "default": "<Any default values>",
              },
             "field2": {
                "type": "double",
                "default": "<Any default values>",
              },
              . . .
          },
          # The price can be per unit of data or as continuous basis such as monthly
          # subscription. The buyer can request for either type of access.
          "price": <Price in STORE per unit of the data>,
          "subscriptionPrice": <Subscription price, if this is what the buyer wants>,
          "subscriptionDuration": <WEEKLY|MONTHLY|ETC.>,
          "metadata": {<Other metadata such as ToC, etc., for this tier>},
      },
      . . .
    },

    # Metering the resource usage. Each of the resources priced above can be
    # metered get the "current usage". The following expressions can be called at any
    # time to know the current resource usage.
    meter_resources: {
      # Memory is provided by Validators to run the app. Unlike other resources
      # listed above, the memory is reclaimed after the completion of the app
      execution,
      # so memory usage is not accumulated over time. Running this expression returns
      # the current memory usage.
      memory_use: <Expression that shows the current memory usage>,
      # CPU cycle usage is accumulated since the beginning of the time. Every time this
      # expression is called a new, updated value is returned, which shows the total
      # number of CPU cycles used so far.
      cpu_use: <Expression that shows the accumulated CPU cycle usage>,
      bandwidth_use: <Expression that shows the accumulated bandwidth usage in GB>,
      storage_use: <Expression that shows the accumulated storage usage in GB>,
      throughput_use: <Expression that shows the current throughput of the app>

      # Each expression accepts a callback function, which is called when the
      # computation is done. Since it might take a while to compute the usage, the
      # values are provided in the callback.
    },

    # Metering revenue collected from data buyers works similarly. Total STORE
    # earned since the beginning of the year, the beginning of the month, and the
    # beginning of the day are computed. Optionally, the revenue earned for a
    # particular month or a day can be computed as well.
    meter_revenue: {
      revenue_earned: <Expression(period, tier, callback)>
      # The expression accepts a parameter called "period" that computes the revenue
      # earned for the requested period. The period can be any of the following.
      # BOY — Beginning of the current year.
      # BOM — Beginning of the current month.
      # BOD — Beginning of today.
      # YYYY — Total datacoins earned for the specified year. If the year is current,
      #        this is the same as BOY.
      # YYYYMM — Total datacoins earned for the specified month of specified year.
      #          If the year and month are current, this is the same as BOM.
      # YYYYMMDD — Total datacoins earned for the specified date.
      #            If the date is the same as today, this is the same as BOD.
      # The "tier" can be a specific tier name or ALL. If specific tier is specified,
      # the revenue generated for the requested tier is computed.
```

```
      # The expression also accepts a callback and the computed revenue is returned in
      # the callback.
    },

    # Withdraw from app developer’s account, based on the rent agreement
    # discussed above in the “rent” section.
    withdraw: <Expression that withdraws STORE from developer_address and deposits into
                    dWorker_address.>
    },

  # Contract hash is the SHA256 of the body of the contract. The hash ensures that
  # the contract body is unaltered by any malicious actors. See signatures below.
  contract_hash: <SHA256(body)>,

  # The contract is signed by the required number of dWorkers and app developers.
  # On the app developers’ side it can be a single representative or multiple
  # people signing the contract. BLS aggregate signature is assumed. This may require
  # distributed key generation (DKG) so all signers get their respective key parts.
  # The signature is produced on contract_hash above. This ensures that the contract
  # details are agreed upon by signers and the contract is not tampered by one or more
  # malicious actors.
  bls_signature_aggregate: <BLS aggregate signature of all signers>
}
```

Listing 1 — Smart contract sample for the rent model

For efficiency purposes, the computations on the resource usage and revenue realization are carried out once a day. While the metering described above works on a continuous basis, computations based on the metered data are triggered once a day. In other words, the smart contract between the app developers, dWorkers, and optionally with end users is executed once every day.

Smart contract for the revenue share model

Listing 2 shows a sample smart contract for the revenue sharing model.

```
{
  # Body of the contract.
  body: {
    # Datacoin name or symbol.
    name: <name of the datacoin>,

    # Version number of the contract. The contract may go through multiple revisions
    # in its lifetime.
    version: <version number>,     # Ex: “1.0.1”.

    # Developer’s wallet address is recorded in the contract. The payments from
    # data buyers is deposited to this address and payments to dWorkers is
    # withdrawn from this address.
```

```
developer_address: <App developer's wallet address>,

dWorker_address: <A wallet address representing the dWorker group>,

# Initial supply of datacoins. This is submitted in the app proposal, but
# dWorkers need to "approve" it to get into the smart contract.
initial_supply: <Approved supply of datacoins>,  # Ex: 1000000000 (1B).

# If premining is allowed so that app developers use the premined tokens to fund
# the app development. The premined tokens are available right away. This is
# optional. If not specified or 0, no premining is allowed.
premine_percentage: <% of initial_supply that can be premined>, # Ex: 1.00

# Annual inflation, if the datacoins are emitted annually based on initial_supply.
# This is optional.
annual_inflation: <% inflation>, # Ex: 2.00

# Revenue sharing agreement between the 3 parties -- app developers, dWorkers,
# and optionally, app end users. The total should be 100%.
# Individual percentages can be numbers like, 30.00 for 30% or "expressions"
# that compute the % dynamically based on whatever factors that app developers
# and dWorkers agreed upon. If expressions are used, they return a number.

revenue_sharing: {
  dWorker_share: <% share or expression>,   # Ex: 30.00.
  developer_share: <% share or expression>, # Ex: 60.00
  # User share is optional. The contract only sets aside the defined % of
  # the revenue for end users. The exact mechanism of sharing that revenue
  # with individual users is outside of the scope of the smart contract.
  # It is up to the app how this distribution is done.
  user_share: <% share or expression>,      # Ex: 10.00.
  # If expressions are used, a Javascript function can be supplied for any of the
  # above. For example:
  # function() {
  #     let percentage_share = 0;
  #     // Logic to compute the % share. Any data in the smart contract can be
  #     // used to compute the % share.
  #     return percentage_share;
  # }
},

# The revenue balance shows the current balance of datacoins owned by
# dWorkers, app developers, and optionally by end users. Note that this balance
# for dWorkers shows the balance for the entire node group and doesn't deal
# with how much revenue individual dWorker node will be receiving. That computation
# is done similar to how the block rewards are shared in the Storecoin
# settlement layer.
revenue_balance: {
  # See transfer below on how these balance amounts are updated.
  dWorker_balance: <number of datacoins owned by the dWorkers>,
  developer_balance: <number of datacoins owned by app developers>,
  user_balance: <number of datacoins owned by end users>
},

# The annual compute resources guaranteed by dWorkers for the app. These numbers
are
# for running the app for 1 year. The numbers below can be “Infinity” to provide
# unlimited resources to the app. Even if the resources are unlimited, they are
still
# metered, so that dWorkers know exactly how much resource has been used by the
app.
compute_resources: {
  memory_in_gb: <Runtime memory in GB>, # Ex: 256.00
  # A “cycle” is CPU usage for 1 second of app instance execution.
  cpu_cycles: <Annual CPU cycles reserved>,
  bandwidth_in_gb: <Annual bandwidth available in GB>,
  storage_in_gb: <Annual storage guaranteed in GB>,
  throughput: <Number of concurrent app instances supported in transactions/second>
},

# Metering the resource usage. Each of the resources guaranteed above can be
metered
# get the “current usage”. The following expressions can be called at any time
# to know the current resource usage.
meter_resources: {
  # Memory is provided by Validators to run the app. Unlike other resources
  # listed above, the memory is reclaimed after the completion of the app
  execution,
  # so memory usage is not accumulated over time. Running this expression returns
  # the current memory usage.
  memory_use: <Expression that shows the current memory usage>,
  # CPU cycle usage is accumulated since the beginning of the time. Every time this
  # expression is called a new, updated value is returned, which shows the total
  # number of CPU cycles used so far.
  cpu_use: <Expression that shows the accumulated CPU cycle usage>,
  bandwidth_use: <Expression that shows the accumulated bandwidth usage in GB>,
  storage_use: <Expression that shows the accumulated storage usage in GB>,
  throughput_use: <Expression that shows the current throughput of the app>

  # Each expression accepts a callback function, which is called when the
  # computation is done. Since it might take a while to compute the usage, the
  # values are provided in the callback.
},
# Metering revenue from datacoins works similarly. Total datacoins earned since
# the beginning of the year, the beginning of the month, and the beginning of
# the day are computed. Optionally, the datacoins earned for a particular month or
a day
# can be computed as well.
meter_revenue: {
  datacoins_earned: <Expression>
```

```
      # The expression accepts a parameter called “period” that computes the datacoins
      # earned for the requested period. The period can be any of the following.
      # BOY — Beginning of the current year.
      # BOM — Beginning of the current month.
      # BOD — Beginning of today.
      # YYYY — Total datacoins earned for the specified year. If the year is current,
      #        this is the same as BOY.
      # YYYYMM — Total datacoins earned for the specified month of specified year.
      #          If the year and month are current, this is the same as BOM.
      # YYYYMMDD — Total datacoins earned for the specified date.
      #            If the date is the same as today, this is the same as BOD.
      # The expression also accepts a callback and the computed revenue is returned in
      # the callback.
      },

    # The transfer method computes the revenue that each party (dWorkers, app
    # developers, and optionally end users) earns based on revenue_sharing
    # structure described above. When this method is called, the revenue_balance
    # data structure is updated with the latest balance.
    transfer: <Expression that computes the revenue balances>
  },

  # Contract hash is the SHA256 of the body of the contract. The hash ensures that
  # the contract body is unaltered by any malicious actors. See signatures below.
  contract_hash: <SHA256(body)>,

  # The contract is signed by the required number of dWorkers and app developers.
  # On the app developers’ side it can be a single representative or multiple
  # people signing the contract. BLS aggregate signature is assumed. This may require
  # distributed key generation (DKG) so all signers get their respective key parts.
  # The signature is produced on contract_hash above. This ensures that the contract
  # details are agreed upon by signers and the contract is not tampered by one or more
  # malicious actors.
  bls_signature_aggregate: <BLS aggregate signature of all signers>
}
```

Listing 2 — Smart contract sample for revenue share model.

The smart contract can be run at any time on the Storecoin Platform to compute the current resource usage or the datacoins earned so far. It can be invoked by any of the dWorkers or the respective app developers (signers described in listing 1.) Although the contract itself can be invoked any number of times, it is recommended that it is invoked once a day in order to compute the respective balances and transfer the datacoin revenue among the parties involved in the contract.

Conclusion

Storecoin Platform uses the smart contract approach to capture the details of the contract between the dWorkers, app developers, and optionally, the end users of the app. The app revenue is generated from the demand for the app data by data buyers. Two models are in place — the rent model and the revenue share model. The rent model allows app developers to rent compute resources from dWorkers. dWorkers are paid in STORE. In the revenue share model, app developers share the revenue with dWorkers to compensate them for the compute resources they provide. The smart contract approach automates execution of contract steps in a secure manner. It also ensures that no party can maliciously tamper the specifics of the contract. This is done by requiring dWorkers and app developers to sign the contract, so any malicious changes will be flagged and the contract execution fails.