Holo Fuel Model

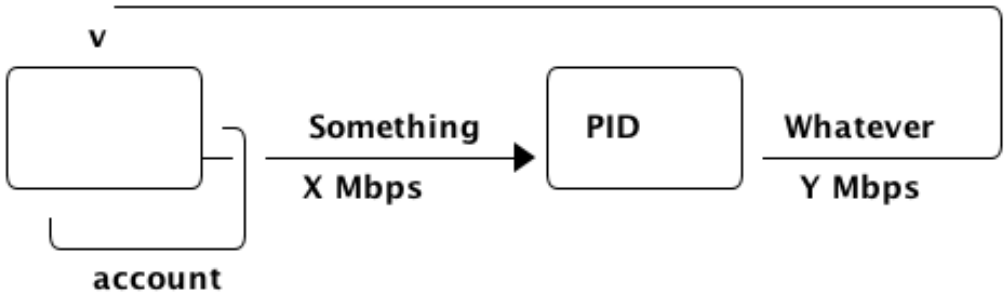
Perry Kundert

July 19, 2018

Contents

1 Holo Fuel

A value-stable wealth-backed cryptocurrency platform, where each unit is defined in terms of a basked of computational resources, operating in a powerful decentralized verification environment.



1.1 The Computational Resources Basket

One Holo Fuel is defined as being able to purchase 1 week of:

Amount	Units	Commodity	Description
1	x	Holo	Inclusion in the Holo system
1	Core	CPU	A processing core
1	GB	RAM	Processor memory
1	TB	Net	Internet bandwidth
25	GB	Data	Persistent storage (DHT/DB/file)

This is roughly equivalent to the 2018 price of cloud hosting (eg. a $5/month Droplet on Digital Ocean), with a premium for inclusion in the Holo system.

1.1.1 Resource Price Stability

There are many detailed requirements for each of these commodities, which may be required for certain Holochain applications; CPU flags (eg. AVX-512, cache size, ...), RAM (GB/s bandwidth), HDD (time to first byte, random/sequential I/O bandwidth), Internet (bandwidth/latency to various Internet backbone routers).

The relative distribution of these features will change over time; RAM becomes faster, CPU cores more powerful. The definition of a typical unit of these commodities changes; as Moore's law decreases the price, the specifications of the typical computer also improve.

For each metric, the price of service on the median Holo Host node will be used; 1/2 will be below (weaker, priced at a discount), 1/2 above (more powerful, priced at a premium). This will offset the natural inflationary nature of Holo Fuel, if we defined it in terms of fixed 2018 computational resources.

1.2 Commodity Price Discovery

Value stabilization requires knowledge of the current prices of each commodity in the currency's valuation basket, ideally denominated in the currency itself. If these commodities are traded within the cryptocurrency implementation, then we can directly discover them on a distributed basis. If outside commodity prices are used, then each independent actor computing the control loop must either reach consensus on the price history (as collected from external sources, such as Distributed Oracles), or trust a separate module to do so. In Holo Fuel, we host the sale of Holo Host services to dApp owners, so we know the historical prices.

When a history of Holo Hosting service prices is available, Linear Regression can be used to discover the average fixed and variable component costs included in the prices, and therefore the current commodity basket price.

1.2.1 Recovering Fixed and Component Costs

```
%matplotlib inline
from __future__ import absolute_import, print_function, division
try:
  from future_builtins import zip, map # Use Python 3 "lazy" zip, map
except ImportError:
  pass
import matplotlib.pyplot as plt
import numpy        # .random, ...
from sklearn import linear_model
import math
import collections

commodity_t         = collections.namedtuple(
    'Commodity', [
'units',
'quality',
'FOB',
    ] )
commodities         = {
    'holo':         commodity_t( "Host",    "",           "" ),
    'cpu':          commodity_t( "Core",    "Median",     "Average" ),
    'ram':          commodity_t( "GB",      "Median",     "Average" ),
    'net':          commodity_t( "TB",      "Median",     "Average" ),
    'data':         commodity_t( "TB",      "Median",     "Average" ),
```

```
}
trade_t             = collections.namedtuple(
    'Trade', [
'security',
'price',
'time',
'amount',
'agent',
      ] )
# The basket represents the computational resource needs of a typical Holochain dApp's "interface"
# Zome.  A small dual-core Holo Host (ie. on a home Internet connection) could perhaps expect to run
# 5 of these; a quad-core / 8-thread perhaps 20.
basket            = {
    # Commodity     Amount
    'holo':        1.  , # Host
    'cpu':          .25, # Core
    'ram':          .25, # GB
    'net':          .1 , # TB
    'data':         .1 , # TB
}
# To simulate initial pricing, lets start with an estimate of proportion of basket value represented
# by each amount of the basket's commodities.  Keep it simple; all are roughly equally weighted.
price_mean          = 1.000                 # target price: 10.0 Holo Fuel == 1 basket
price_sigma         = price_mean / 10       #  difference allowed; about +/- 10% of target
prices              = { k: ( price_sigma * numpy.random.randn() + price_mean ) / len( basket ) / basket[k]
      for k in basket }
'''
[ "%10s: $%5.2f / %s" %
 + [ "Basket cost: $%5.2f" % ( sum( prices[k] * basket[k] for k in basket )) ]
'''
out = "\n".join
"""\
'Commodity   HOT$ Price'
'---------   ----------------------'
%s
%s
%s
%s
%s
""" % tuple( "%-10s %5.2f / %-5s / mo." % ( k, prices[k], commodities[k].units ) for k in basket )
```

Once we have the currency's underlying commodity basket, lets simulate a sequence of trades of various amounts of these commodities. We will not know the exact costs of each commodity used to compute the price, or how much is "baseline" Holo system premium; costs not directly related to the amounts of commodities (eg. per-transaction fixed costs, etc.)

Lets see if we can recover the approximate Holo baseline and per-commodity costs from a sequence of trades. Create some trades of baseline Holo + random amounts of commodities 5 +/- 2, adjusted by a random amount (ie. 'holo' always equals 1 unit).

```
amounts_mean        = 5
amounts_sigma       = 2
error_sigma         = 0.05 # +/- 5% variance in bids vs. price
trades              = []
number              = 1000
for _ in range( number ):
    amounts         = { k: 1 if k == 'holo' else max( 0, int( amounts_sigma * numpy.random.randn() + amounts_mean )) for k in ba
    price           = baseline + sum( amounts[k] * prices[k] for k in amounts )
    error           = price * error_sigma * numpy.random.randn()
    bid             = price + error
    trades.append( dict( bid = bid, price = price, error = error, amounts = amounts ))
"""\
'HOT$   calc. err.  Requirements'
'-----  ----- ----  ------------'
%s
%s
```

```
%s
%s
%s
""" % tuple( "%5.2f (%5.2f%+5.2f) for %s" % (
t['bid'],
t['price'],
t['error'],
", ".join( "%5.2f %s %s" % ( v * basket[k], k, commodities[k].units ) for k,v in t['amounts'].items() ))
     for t in trades[:5] )
```

Compute a linear regression over the trades, to try to recover an estimate of the baseline + commodity prices.

```
items               = [ [ t['amounts'][k] for k in basket ] for t in trades ]
bids                = [ t['bid'] for t in trades ]
regression          = linear_model.LinearRegression( fit_intercept=False, normalize=False )
regression.fit( items, bids )
items_pred          = { k: regression.predict( [ int( k == k2 ) for k2 in basket ] ) for k in basket }

# "Score(R^2): %r" % ( regression.score( items, bids ) )
"""\
'Commodity  HOT$  Error in linear regression
%s
%s
%s
%s
%s

""" % tuple( "%-10s %5.2f %+5.2f%%" % ( k, items_pred[k], ( items_pred[k] - prices[k] ) * 100 / prices[k] )
for k in basket )
```