Holo Fuel Reserve Account Design and Modelling

Perry Kundert

October 19, 2018

Contents

Abstract

The Holo Reserves are a primary method of purchasing Holo Fuel for Hosting services, and is available for Hosts only to redeem Holo Fuel for cash in various currencies. Other Holo Fuel account holders may buy via the Reserves, and may buy/sell via other exchanges, but the Reserve's LIFO tranches are available only to Holo Fuel accounts associated with known Holo Hosts, and only for Holo fuel earned for hosting services.

Holo Fuel credits redeemable on Reserves for Hosting are purchased at a certain cost, and later redeemed for that same amount by Hosts after these services are delivered. Therefore, the purchase price must be palatable for redemption later by at least some Hosts. Of course, Holo dApp Owners are free to purchase Holo Fuel on exchanges at lower prices, and Hosts can also cash out via exchanges at market prices.

The Holo Fuel to currency sale price is also controlled to adjust net currency in/outflows, both to adjust for changes in relative currency valuation, and to balance the proportion of Reserves in each currency to match the desired Host cash-out currencies.

Models are proposed to implement a sound foundation of wealth in terms of Holo Hosting services, savings balances, and healthy monetary velocity in trade transactions for goods and services. Early steady deflation (increase in Holo fuel unit value in real terms) will be managed as the scale and velocity of the Holo fuel economy increases, leading ultimately to an engineered equilibrium value stability plateau where Holo fuel becomes extremely useful as general "money".

1 Holo fuel: Sound Money

The intended utilisation of Holo fuel as money to trade in goods and services (Holo Hosting, etc.) requires that it have at least the features required of a Medium of Exchange. Eventually, after an initial period of price discovery and growth, it must also become a Measure/Store of Value.

1.1 Medium of Exchange

Holo fuel inherits many of these required features from its Holochain and Mutual Credit foundations.

- Valuation vs. common assets
- Common and accessible
- Constant utility
- Low cost of preservation
- Transportability
- Divisibility
- High market value in relation to volume and weight
- Recognisability
- Resistance to counterfeiting

During the initial price discovery phase, the existing pool of 177e9 HOT ERC20 tokens will establish an equilibrium supply/demand price vs. other Cryptocurrencies supplying similar utility. We expect a period of neutrality or inflation (lowering valuation) as savers/sellers (initial investors and HODLers, Holo Hosts) seek equilibrium of Holo fuel supply with spenders/buyers (Holo dApp owners, other accumulating savers, and a small proportion of monetary trade users).

As the superior utility of Holochain based Mutual Credit currencies is established vs. existing Cryptocurrency offerings (vastly superior scalability, ease of use and security), a period of deflation (increasing value) is expected. Increases in value are moderated by increasing selling (profit-taking) by early investors. However, a target valuation for value-stability begins to form, in preparation for the long-term primary utilization of Holo fuel as general "money".

1.2 Measure and Store of Value

To serve as a measure/store of value, a medium of exchange, be it a good or signal, needs to have constant inherent value of its own or it must be firmly linked to a definite basket of goods and services. It should have constant intrinsic value and stable purchasing power; otherwise, it will be chased out of the market by "worse" money: all else being equal, people will generally choose to **save** units of money that retain or increase in value (neutral/deflationary), and will **spend** money that declines in value (inflationary).

Long-term monetary value stability in a dynamic economy of mutual-credit currencies requires the extension and spending (and potentially later retraction/repayment) of credit lines (similar to "minting" and "burning" in a token currency). Increasing savings or monetary velocity demands increases in the money supply to avoid deflation; spending of savings or a reduction in economic trade activities requires reduction of the money supply. This is accomplished by offering credit

lines and/or incentivizing their usage, and later withdrawing credit lines and/or incentivizing their repayment.

Credit extended in terms of money that increases in unit value (deflation) becomes harder to pay back over time, so it is unlikely (and unhealthy) for people to deploy credit lines unless it is both expected and **highly** likely that deflation will only persist for a short term before recovery. Conversely, money that loses value (inflation) is not useful for savings, and persistent (or expected) inflation will cause people to transfer their savings to more sound money.

1.2.1 Controlled Withdrawal of Supply To Counter Inflation

Large-scale withdrawal of units of money that are not backed by actual wealth leads to a crash of the monetary system; there is no mechanism for a stable "unwinding" of non wealth-backed money. As units sold exhaust buy orders, prices drop as demand for the money is satisfied. Existing credit lines can be reduced, and/or deployed credit lines repaid with the inflated (lower priced) units of currency (on the assumption that unit values will be restored to "normal"). As inflation persists, feedback control loops continue to restrict credit lines until unit prices respond to the tightening supply, which is being soaked up by creditors paying back their lines at discounted current unit rates.

But, what if credit lines are exhausted before the currency responds; demand is simply lower than the amount of units in circulation available for sale? The 177e9 HOT minted are convertible 1-to-1 into Holo fuel, and if the amount selling exceeds demand for Holo services and general trade, there is no lower bound on its per-unit valuation vs. currencies w/ similar utility. Without the ability to effectively limit supply to existing demand, prices will collapse. The larger the uncontrolled supply vs. the controllable supply issued via credit, the greater the probability is of exhausting available credit withdrawal capacity.

How can we reduce the relative size of the uncontrolled, "minted" supply in circulation?

1. Increasing the size of the Holo fuel economy

   The obvious way is to dramatically increase the scale of the economy. Long term savings is restricted to a relatively small portion of the economy – those choosing to save disposable wealth in the form of Cryptocurrency.

   The **much** larger short-term savings and spending economy is closed off to any currency that has high volatility – is not strongly value-stable. A strongly value-stable currency that doesn't have the fee and scale barriers of existing Cryptocurrencies would be very attractive. The bulk of the global population labours under spectacularly onerous regimes (pay-day loans, high-fee mobile-phone company currencies, etc.), and there are **trillions** of dollars actively seeking a new home in the short-term and daily spending market.

   The determining factors limiting uptake are:

   - Strong value stability
   - Low fees in the range of retail transactions (USD$10-10,000)
   - Extremely secure/resilient key revocation and recovery (community based)

2. Directly Reducing Non-Wealth-Backed "minted" Units

   Like a share buy-back scheme, a proportion of the Holo fuel transaction fees should redeem the Holo fuel units initially issued by the Holo organization to raise initial capital. This, strangely, depends on 1 – without dramatically increasing the monetary velocity of Holo fuel,

to (at least) on the order of some multiples of the total outstanding stock of non-wealth-backed Holo fuel per year, the transaction fees will be inadequate to make a dent in the originally issued ERC20 token based Holo fuel.

As with taxes, raising fees may actually lower fee income. This is the effect, I believe, of a blanket 1% fee structure – it makes micro-transactions extremely compelling vs. any competing platform, but completely eliminates Holo fuel from consideration in that **vastly** larger retail savings and transaction pool. In my opinion, .1% of a Trillion USD$ retail market is much better than 1% of . . . no part of that market.

Fees are discussed below.

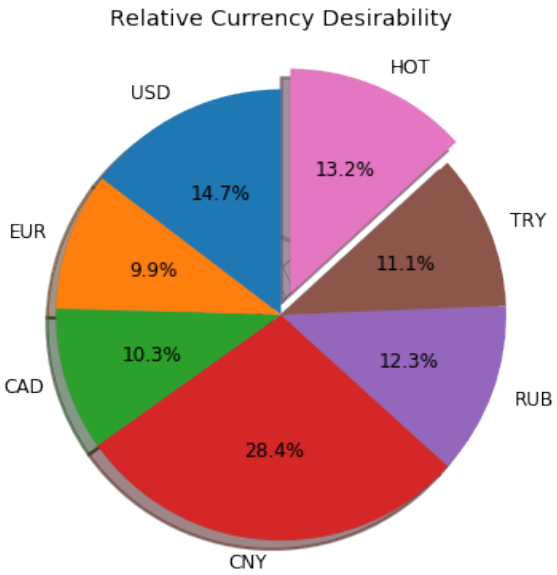
2 Host Currency Preference and Minimum Cash-Out Value

Each Host sets their preference for redemption currencies, and an exchange rate minimum for one of them; the others will be deduced, because each of their exchange rates to Holo Fuel are known.

Lets select a few currencies, with varying levels of desirability for Holo Hosts to cash out with:


```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

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 bisect
import collections
import json
import logging
import math
import numpy as np
import random
import scipy # stats.zscore, stats.norm.cdf, ...
import sys
import time

from datetime import datetime, timedelta
from sklearn import linear_model

import matplotlib
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]  = (6,3)
plt.rcParams["font.size"]       = 6

#
# Simulation parameters
#
sim_duration                    = 14 * trading.day


logging.basicConfig(
    stream=sys.stdout,
    level=logging.WARNING, # INFO, DEBUG,
    datefmt='%Y-%m-%d %H:%M:%S',
    format='%(asctime)s.%(msecs).03d %(threadName)10.10s %(name)-8.8s %(levelname)-8.8s %(funcName)-10.10s %(message)s' )

from holofuel.model import trading
from holofuel.model.reserve_lifo import reserve, reserve_issuing
```

```
def rnd_std_dst( sigma, mean=0, minimum=None, maximum=None ):
    """Random values with mean, in a standard distribution w/ sigma, clipped to given minimum/maximum."""
    val             = sigma * np.random.randn() + mean
    return val if minimum is None and maximum is None else np.clip( val, a_min=minimum, a_max=maximum )

#
# Compute target cu‘‘rrencies, with random distribution of desirabilites (totalling 1.0)
#
currencies                      = [ 'USD', 'EUR', 'CAD', 'CNY', 'RUB', 'TRY', 'HOT' ]
desi_mean,desi_sigma            = 1,    .66     # desirability weighting of various currencies
desirability                    = [ rnd_std_dst( mean=desi_mean, sigma=desi_sigma, minimum=3*len(currencies)/100 ) # ~3% minimum
                                    for _ in range( len( currencies )) ]
desirability                   /= np.sum( desirability ) # normalize sum of probabilites to 1.0

explode                         = [ .1 if c == 'HOT' else 0 for c in currencies ]
#with plt.xkcd():
fig1,ax1                        = plt.subplots()
ax1.pie( desirability, explode=explode, labels=currencies, autopct='%.1f%%', shadow=True, startangle=90 )
ax1.axis( 'equal' ) # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title( "Relative Currency Desirability" )
plt.show()
```



Obtain some actual exchange rates for a test period:

```
class Rates( object ):
    """Load some exchange rates.  Convert observations:

       [ { "d": "2017-01-03", "FXAUDCAD": { "v": 0.9702 }, ...},
         { "d": "2017-01-04" ... }, ... ]
     to rates:
       { "2017-01-03": { "USD/CAD": 1.29, "CAD/USD": 0.775, "USD/EUR": ... }, "2017-01-04": { ... } }

     """
    def __init__( self, fx_rates ):
        rates_data              = json.loads( open( fx_rates ).read() )
        self.rates              = {}
        for rec in rates_data['observations']:
            d = self.rates[rec["d"]] \
                                = {}
            for c1 in currencies:
```

```
                if c1 != 'CAD' and 'FX'+c1+'CAD' not in rec:
                    continue
                for c2 in set( currencies ) - set([c1]):
                    if c2 != 'CAD' and 'FX'+c2+'CAD' not in rec:
                        continue
                    tocad       = 1 if c1 == 'CAD' else rec['FX'+c1+'CAD']['v'] # eg. FXUSDCAD: 1.3
                    frcad       = 1 if c2 == 'CAD' else rec['FX'+c2+'CAD']['v'] # eg. FXEURCAD: 1.47
                    d[c1+'/'+c2]= tocad / frcad                                 # ==>  USD/EUR: 0.884
        self.days               = sorted( self.rates.keys() )
        logging.debug( "Rates: %d days loaded: %s, ...", len( self.days ), json.dumps( self.rates[self.days[0]], indent=4 ))

    def seconds_to_day( self, seconds ):
        """Given a 'now' offset in seconds from 0, compute the current day, relative to the first available rate data point"""
        dt                      = datetime.strptime( self.days[0], '%Y-%m-%d' )
        dt                     += timedelta( seconds=seconds )
        return dt.strftime( '%Y-%m-%d' )

    def exchange( self, day, fr, to ): # "2017-01-17", 'EUR', 'USD'
        """Finds the exchange rate for a day near to the given YYYY-MM-DD"""
        if fr == to:
            return 1.0
        i                       = bisect.bisect_left( self.days, day )
        if i >= len( self.days ):
            i                  -= 1
        #print( json.dumps( self.rates[self.days[i]], indent=4 ))
        try:
            return self.rates[self.days[i]][fr+'/'+to]
        except:
            return math.nan

rates                           = Rates( "static/data/FX_RATES_DAILY-sd-2017-01-03.json" )

[ [ 'Day', 'From', 'To', 'Exchange' ],  None ] \
+ [ [ d, fr, to, "%5.3f" % ( rates.exchange( d, fr, to ) ) ]
    for d in [ '2017-01-04', '2018-09-20' ]
    for fr,to in [['EUR','USD'],['USD','EUR'],['CAD','EUR'],['CAD','USD']] ]
```


Day	From	To	Exchange
2017-01-04	EUR	USD	1.046
2017-01-04	USD	EUR	0.956
2017-01-04	CAD	EUR	0.718
2017-01-04	CAD	USD	0.751
2018-09-20	EUR	USD	1.176
2018-09-20	USD	EUR	0.850
2018-09-20	CAD	EUR	0.659
2018-09-20	CAD	USD	0.775

Each Host can specify 0 or more preferred redemption currencies and rates. Only 1 target Fiat currency rate is allowed, because the exchange rates between currencies are deduced by the inflow/outflow equilibrium through the Reserve accounts. Until HOT floats, no exchange rate is supported; it is fixed at 1 HOT == 1 Holo Fuel.

```
class Host( trading.agent ):

    capacity_multiplier         = 1                             # change to alter all Host's .cores/.capacity

    def __init__( self, redemption,
            quanta = 1 * trading.hour,
            capacity_mean = 1.0, capacity_sigma = 0.2,          # 'holo' host-month capacity
            cores = 1000, **kwds ):                             # Represents this many host-cores

        """Support 0 or 1 specified exchange rate, deducing all others.  Filter out currencies not desired
        (target rate is Falsey).
```

```
        redemption: {
            "CAD": .50,
            "USD": True,
            "CNY": False, # Filtered out
            "EUR": True,
            "HOT": True
        }

        """
        super( Host, self ).__init__( quanta=quanta, **kwds )
        self.redemption         = { c: redemption[c]
                                    for c in redemption
                                    if redemption[c] }

        assert 0 <= sum( type( r ) in (int,float) for c,r in self.redemption.items() ) <= 1, \
            "A maximum of one target redemption is allowed; %s supplied" % (
                ', '.join( '%s: %f' % ( c, r )
                           for c,r in self.redemption.items()
                           if type( r ) in (int,float) ))

        self._cores             = cores
        self.capacity_mean      = capacity_mean
        self.capacity_sigma     = capacity_sigma
        self._capacity          = rnd_std_dst( mean=self.capacity_mean, sigma=self.capacity_sigma, minimum=0 )

    def sells_to( self, another ):
        """Hosts only sell (redeem) Holo fuel to reserves in this simulation. """
allowed = isinstance( another, reserve ) and super( Host, self ).sells_to( another )
        logging.info( "%s buys from %s: %s", self, another, allowed )
        return allowed

    @property
    def capacity( self ):
        return self._capacity * self.capacity_multiplier

    @property
    def cores( self ):
        return self._cores * self.capacity_multiplier

    def redemption_rate( self, day, curr ):
        """Computes the target redemption rate in the specified currency, or Falsey (0/None/False) if not
        desired.  If a currency is desired, but no minimum cash-out rate is specified (indicating
        that "market" rates are desired), returns True."""
        if curr not in self.redemption:
            return False
        if curr == 'HOT':
            return 1.0
        # find a specified currency w/ a minimum rate specified
        for curr_exch,rate_min in self.redemption.items():
            if type( rate_min ) is not bool: # could be int,float, a numpy type
                # An exchange rate minimum was specified!  Compute the target currency's rate vs. that
                # rate using that day's (in "YYYY-MM-DD") exchange rate.  For example, if the exch ==
                # 'USD' and the target is (say) rate == 0.50, and we're asking for 'CAD' and the day's
                # exchange rate is 1.29, we'll return 0.50 * 1.20 == 0.645
                rate_exch               = rates.exchange( day, fr=curr_exch, to=curr )
                rate_redeem             = rate_min * rate_exch
                if math.isnan( rate_exch ) or math.isnan( rate_redeem ):
                    logging.warning( "For %s on %s, minimum: %s, %s/%s exchange rate: %s" % (
                        curr, day, rate_min, curr_exch, curr, rate_exch ))
                return rate_redeem
        # No target currency w/ minimum rate: "market" rates are desired
        return True

    def run( self, exch, now=None ):
        """Runs the Host trading.actor's needs/targets and issues trades as required to get self.targets
        satisfied.  Hosts will manually sell their holdings when Reserve prices hit their targets; so, list
        our holdings at our target price.
```

```
        We will make these trades "old"; they are basically standing orders, and should get the best
        price available from the Reserves.  Since the holofuel.model.trading.exchange gives the
        spread to the trade carrying the most risk (the "market" order, or the oldest limit order),
        this will ensure that the Host always get the benefit of the best bid (buy) price the
        Reserve (or other party) has to offer.  """

        if not super( Host, self ).run( exch=exch, now=now ):
            return False # Quanta not yet satisfied; do nothing
        # Enabling this kills reserves_issuing... Fix it so Reserves will buy at their bid price,
        # regardless of what Hosts are willing to sell at.  Right now, the "oldest" of the bid/ask gets the better
        # price (the spread).  Perhaps always make the time on the Reserve bid with a far future timestamp?
        #return True # NO-OP for now.
        if self.assets.get( 'Holofuel' ): # not None/0:
            day                 = rates.seconds_to_day( seconds=now )
            amount              = self.assets['Holofuel']
            price               = self.redemption_rate( day=day, curr=exch.currency )
            if price:
                # This Host has specified a redemption rate for this currency (eg. USD); if True, this means market rates
                if price is True:
                    price       = None
                logging.info( "{} selling {} Holo fuel at {} at exch rate of day {}".format( str( self ), amount, price, day ))
                exch.sell( agent=self, amount=amount, price=price, security='Holofuel', now=0, update=True )

#
# Compute a number of Host w/ varying numbers of desired currencies and target exchange rates
#
host_count                      = 25
rate_mean,rate_sigma            = 0.50, 0.25    # variance in minimum rates of exchange (CAD)
curr_mean,curr_sigma            = 3,    2       # number of currencies selected
hosts                           = []

for h in range( host_count ):
    # select between 0 and all currencies as candidates for redemption, with the random choice of each
    # currency weighted by its relative desirability
    curr_cnt                    = max( 0, min( len( currencies ), int( rnd_std_dst( mean=curr_mean, sigma=curr_sigma ))))
    redemption                  = { curr: True
                                    for curr in np.random.choice( a=currencies, size=curr_cnt, replace=False, p=desirability ) }
    # Choose an exchange rate for one Fiat currency (in CAD$ terms)
    fiat                        = set( redemption ) - set( [ 'HOT' ] )
    rate_num                    = 1
    rate_cad                    = rnd_std_dst( mean=rate_mean, sigma=rate_sigma, minimum=0 ) # may be 0 ==> no desired rate ("ma

    if fiat and rate_cad:
        for curr in np.random.choice( a=list( fiat ), size=min( rate_num, len( fiat )), replace=False ):
            redemption[curr]    = rate_cad * rates.exchange( rates.days[0], 'CAD', curr )
    hosts.append( Host( identity = "Host {}".format( h ), redemption = redemption ))
    logging.info( "CAD exch: %6.4f, target Fiat %r == %r %s; Holdings: %r",
        rate_cad, fiat, hosts[-1].redemption, "" if rate_cad else "==> market rates",
        hosts[-1].assets )


#
# See if we can recover a median, mean and std.dev. for each cash-out currency.
#
def currency_statistics( hosts, day, curr ):
    """For a currency 'curr' on a day, compute the Hosts desiring that currency, and the statistical
    distribution of their cash-out minimum.

    """
    curr_stats                  = {}
    # Ignore bad, Falsey (False/0 == not desired), or -'ve (invalid) exchange rates
    sel                         = []
    for h in hosts:
        r                       = h.redemption_rate( day, curr )
        if not math.isnan( r ) and r and r > 0:
            sel.append( r )
    if not sel:
        return curr_stats # leave empty (Falsey) if no cash-out currencies selected
```

```
    curr_stats['selected']      = sel # contains desired exch. rate, or True (for "market")
    curr_stats['minimums']      = sorted( x for x in sel if type( x ) is not bool )
    mins_cnt                    = len( curr_stats['minimums'] )
    curr_stats['median']        = curr_stats['minimums'][mins_cnt // 2] if mins_cnt else None
    curr_stats['mean']          = np.mean( curr_stats['minimums'] ) if mins_cnt else None
    curr_stats['sd']            = np.std( curr_stats['minimums'] ) if mins_cnt else None
    return curr_stats

stats                           = {}
for curr in currencies:
    stats[curr]                 = currency_statistics( hosts, rates.days[0], curr )
    logging.debug( "Minimum cash-out %s", curr + ': ' + ', '.join( "%7.4f" % r for r in stats[curr]['minimums'] ))

[ [ '', '', '%r/ea +/-%r' % ( curr_mean, curr_sigma ), 'Rate' ],
  [ 'Currency', '% Weight', '% Selected', 'Mean', 'Median', 'Std.Dev' ],
  None ] \
+ [ [ curr,
      "%.1f" % ( desi * 100 ),
      len( stats[curr]['minimums'] ) * 100.0 / host_count,
      "%.4f" % ( stats[curr]['mean'] or 0 ),
      "%.4f" % ( stats[curr]['median'] or 0 ),
      "%.4f" % ( stats[curr]['sd'] or 0 ) ]
    for curr,desi in zip( currencies, desirability ) ]
```

Currency	% Weight	3/ea +/-2 % Selected	Rate Mean	Median	Std.Dev
USD	14.7	48.0	0.3288	0.3343	0.1429
EUR	9.9	28.0	0.3450	0.3297	0.1534
CAD	10.3	28.0	0.4116	0.4394	0.1720
CNY	28.4	56.0	2.1614	2.3867	0.8589
RUB	12.3	32.0	18.7792	19.8821	7.3146
TRY	11.1	32.0	1.4216	1.3179	0.5404
HOT	13.2	20.0	1.0000	1.0000	0.0000

2.1 Host Auto-pilot Pricing

A Host can specify rates to charge for its various computational resources, in Holo Fuel, or it can set "auto-pilot" pricing. The lower the pricing, the higher the expected utilization of the resource vs. the median Host.

Each Host competes for traffic against other Hosts serving the same Holochain hApp. From time to time, the Holo service polls the Hosts capable of serving an hApp, and groups them into tranches of comparable quality based on price. A proportion of the hApp's traffic will be assigned to each tranche; more to lower-priced tranches, less to the more costly.

Thus, over time the Hosts' pricing decisions will be reflected in the average utilization for the resource. This could be computed over days, not hours, to account for cyclical (day/night) shifts in utilization. Or, it could be computed on a shorter cycle such as every 10 minutes, to allow the auto-pilot to be used to adjust utilization more promptly.

To support real-time utilization modulation, for example increasing the price of Network bandwidth to reduce utilization when the owner is using a streaming video services like Netflix, would also require the Holo system supporting the hApp to poll its Host resources for pricing more rapidly; at the Nyquist rate; 2x the frequency of change of the signal.

Each Host sets its own pricing using its own resource utilization PID loop. These new prices must be scanned from public DHT entries at 2x the rate at which the Holo system wishes to respond to Host rate changes. Previous Host resource pricing persists until it is scanned and re-tranched w/ its new pricing. A cycle of minutes may not be inappropriate for this, and may also be considered part of the Host "liveness" testing.

2.1.1 Increasing hApp Hosting Prices Attracts New Hosts

As a Host wishing to maximize revenue per unit of Compute, I want to host hApps that pay well. Each Holo hApp knows what its median and average hosting utilization and prices have been across all resources, and this information is published; both facts are necessary to intelligently select hApps that fit the desired resource and income profile of the Host.

Hosts will survey the hApps available from time to time, disabling and eventually ejecting low-paying (probably over-provisioned) hApps in favour of higher-paying (possibly under-provisioned) ones. This eventually frees up the storage and other resources used by the old hApp; once the Host is no longer represented in the hApps tranches, it can power down and delete the hApps' resources.

Each hApp uses various resources (eg. Network bandwidth, CPU power, RAM, Storage) at differing rates. One or more hApps will be ejected only if the replacement hApp(s) fill all of the available Host resources more profitably than the old set.

Equilibrium is reached when hApps are provisioned across the Hosting network with all Hosts' resource utilization more or less level (eg. a High CPU Low Storage hApp, next to a Low CPU High Storage hApp), and the median resource cost more or less equal for each hApp, proportional to its average utilization. For example, given two roughly equivalent hApps, one with 100x more client utilization than the other; the Holo Host pricing system should ensure that roughly 100x more Hosts are hosting the hApp, and that the aggregate Hosting costs to the larger hApp owner are about 100x the costs of the lesser hApp.

2.1.2 hApp/Host Auto-pilot Feedback

If an hApp owner is aware of cyclicality or spikes in its utilization (eg. just before launching an advertising campaign), the owner can even pre-allocate increased resources by temporarily increasing its own hApp Holo service auto-pilot pricing to a higher tier. This increases the amount it is willing to pay for hosting, putting it into contention for installation by Hosts with "hi" (premium) auto-pilot pricing. When the spike actually hits, the hApp owner can restore its own pricing auto-pilot to the normal tier, letting regular Holo price-based levelling distribute the hApp appropriately for the new load.

2.2 Modelling Holo hApp/Host Auto-pilot Pricing

The goal of Holo hApp and Host Auto-pilot pricing is to allow both hApp owners and Hosts to achieve equilibrium pricing within a budget they can afford.

Holo hApp owners have clients to serve, and require Host resources within a certain budget. Hosts have resources to sell, and want to make the most money by hosting the hApps paying the most for those resources.

2.2.1 Holo hApp Host Tranching

A core tenet of Holochain applications is that their state is stored privately in a local chain, and publicly in an eventually consistent DHT. So, in theory, any "read only" client request accessing public data can be served by any Host. The application using Holochain must be resilient to the eventually consistent nature of the underlying datastore. Much of Holo's activity will, however, be the establishment of Holochain proxy instances, which are capable of storing/updating a local chain on behalf of a (web-based) user (the identity's signing keys are held by the client; communication encryption keys are held by the proxies).

The Hosts to provide these services are chosen pseudorandomly from pools of Hosts of like performance and cost, called tranches. The probability of getting any Host is proportional to its desirability (cheapest highest performing hosts first). In aggregate, the average price paid per request is intended to be near the "median" price/performance; a mix of high/low priced and high/low performing Hosts is used. The tranches are dynamically updated based on analysis of the actual request performance and current Host pricing. The mix of Hosts used to service requests is adjusted dynamically based on the hApp owner's current Hosting cost targets; a hApp currently targeting below-market **discount** Hosting costs will get a mixture of Hosts averaging that lower target cost (ie. few **premium** priced Hosts, more **discount** and **market** priced Hosts.)

Each set of tranches is an N-dimensional grid of buckets, with axes denominated in the various ratings for the feature. The 'holo' commodity is simple; a single axis based on transaction response time, as computed by Holo's interfaces on the Host. These buckets are at standard deviation boundaries in the measured data, which is assumed to be more or less normally distributed.

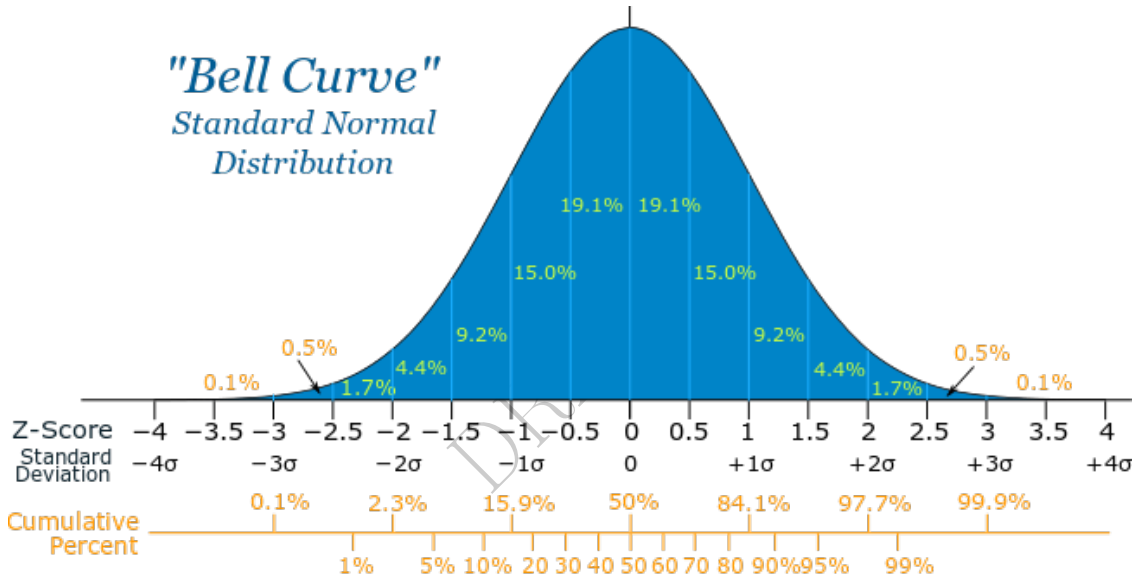


Figure 1: Standard Deviation

The 5 buckets on the "performance" axis contain Hosts which fall in the standard deviation groups vs. the median response time. The 5 buckets on the Holo Host "pricing" axis are selected by each Hosts' dynamically adjusted pricing. The lowest performing $< -2\sigma$ (**bulk** 2%) are not used for serving real-time requests, until their response times to bulk requests moves them out of the `lo lo` group. The 2% **peak** nodes can be reserved for the most performance sensitive requests (eg. CDN like activities). The lowest priced 2% of Hosts are reserved for qualifying "charity" hApps (eg. Wikipedia-likes, etc.), and the highest priced 2% of Hosts are eliminated from consideration; these Hosts would skew the pricing to wildly to include in the tranche (eg. manually-priced Hosts demanding USD$1m per unit of resources)

Performance »		2% bulk	28% slow	40% median	28% fast	2% peak
v Price v	Std.Dev.	$< -2\sigma$	-2σ - $-.5\sigma$	$-.5\sigma$ - $+.5\sigma$	$+.5\sigma$ - $+2\sigma$	$> +2\sigma$
2% free	$< -2\sigma$	?				
28% discount	-2σ - $-.5\sigma$	"background"	some requests	most requests	some requests	CDN, web
40% market	$-.5\sigma$ - $+.5\sigma$	API requests		some requests		proxies,
28% premium	$+.5\sigma$ - $+2\sigma$	(none; ignored?)	very few	few request	very few	relay, etc.
2% over	$> +2\sigma$	?				

This arrangement leads to a cooperative feedback loop, allowing both Holo hApps and Hosts to dynamically adjust their pricing:

- A **premium** Host modulates its prices to keep its utilization in a low band, a **discount** Host does so to keep its utilization high. This causes the Holo hApp administration DNA to collect this information from time to time recompute price statistics and standard deviations, and move it directly between **discount**, ..., **premium** tranches in its performance band.
  - Eventually (as its performance reflects its changed utilization), the Holo hApp manager will also migrate it between **bulk**, **slow**, **median**, **fast** and **peak** performance tranches.
- A **discount** hApp adjusts its cost targets to keep its performance in the lower acceptable range, the **premium** hApp adjusts to keep performance in the higher end of the band. It selects random Hosts from various tranches with varying probabilites to achieve its target average Hosting cost.
  - If costs escalate due to overall increasing Hosting costs, its pool of credit supports less runtime. The owner should be informed that they may want to drop to a lower cost target (eg. from **market** to **discount**) to stretch out its hosting account, or put more money in.

Overall, the process of deploying an hApp:

- Holo Host installs the hApp, identifies itself to the hApp manager
- hApp Manager adds it to the lowest performing (probably cheapest) tranche in each resource category
- Holo begins sending requests, collecting signed service logs
- The Host performance tranches are recomputed based on service log resource utilization and response timing
  - Each service response carries the total used Hosts wall-clock duration (units of Holo) and CPU seconds, plus the total (hourly exponential moving average) Storage, RAM and Network utilization and total wall-clock duration of requests served. This allows us to assign a fraction of the total hApp resource utilization to this request, and deduce a price
  - The cost tranche boundaries are recomputed from the latest set of new Host pricing data collected from ongoing DHT scans of all Hosts, and the Hosts are distributed into their new tranches on the cost axis.
- Prices paid per avg. request increase as Hosts move to higher cost tranches, and/or the standard deviation "boundaries" change and the average price in the target tranches increases.

– The hApp manager warns the owner of significant changes in hosting costs, so they can adjust their preferred cost settings.

2.2.2 Variables Affecting Client, hApp, Host and Reserve Interaction

We will simulate a tranche of Hosts selling Holo hosting by the hour. A certain amount of Holo fuel is defined as purchasing a defined basket of the median Holo computational commodities and capabilities required to host a small hApp for 1 month.

The value of Holo fuel in terms of both its defined commodity basket, and its exchange rate to other currencies is going to float to begin with. In addition, its USD$/fuel bid/ask on the Reserve will also fluctuate, within bounds set by the Hosts in the cash-out prices and preferences.

So, what determines the value of a unit of Holo fuel? Existing cryptocurrencies have little to offer except for Utility vs. existing Fiat currencies, and have acquired a market value of about USD$224 billion. Something is attractive about the Holo fuel story, and it is already ranked about the 50th of all available cryptocurrencies, and 3rd in buy vs. sell interest. What factors influence its long term valuation, and what can we do to control them?

1. Security

   Without security, nothing else matters. The Holo DPKI approach is compelling vs. existing "Wallets".

   - Aggregation of multiple accounts under a single identity
   - Key revocation/replacement
   - Recovery from a quorum of secrets distributed to trusted parties or secured separately

2. Utility

   Unless there is significant improved Utility vs. existing Fiat/Cryptocurrency options, there is little incentive to use Holo fuel.

   - Scalability, essentially without bound.
   - Capability to implement full featured distributed applications vs. simple smart contracts

3. Fees

   If fees are high vs. even inferior existing options, uptake will be limited. Since Holo fuel has no scalability limits and little centralized overhead, there is little actual per-transaction infrastructure cost, at scale. The fees should be maximized, however, to allow funding of expansion to supplant inferior currencies and maximize human benefit. What fee structure is likely to maximize short- and long-term fee income?

   - A 1% fee for microtransactions is excellent, and completely out of reach for competing system
   - For larger transactions, it is wildly overpriced vs. all competing services (except credit cards, pay-day loans, etc.)
   - For inter-account transfers, a 1% fee eliminates the possibility of essentially all account maintenance activities (eg. day to day banking and separate savings accounts, etc.)

Capping transaction fees and a sliding scale would address these issues. These should be tuned over time to maximize income and remain competitive:

Fee	Tx Size	Cap	Transaction Type
0%	all	0	Transfers between accounts **within** same DPKI identity
1%	< ~USD$10	~USD$0.01	Microtransaction and small retail
0.1%	< ~USD$10,000	~USD$1	Typical consumer retail transactions
0.01%	>=~USD$10,000	~USD$1,000	Large transactions

Existing Fiat payment systems (eg. Credit/Debit cards, cheques, bank accounts, etc.) and global consensus Cryptocurrency systems cannot do microtransactions or small retail efficiently at all, let alone at a competitive cost (consumers only **think** they can, because all of these costs are hidden from them). Gaining access to the **huge** microtransaction and small retail space is critical to the uptake of Holo fuel, and capping the fees at ~USD$0.01 (1 cent) the Holo fuel equivalent of < ~USD$10.00 purchases will be acceptable, and will cover the vast majority of transactions. The only transactions that will produce reduced Holo system fees are those between ~USD1.00 − 10.00.$Instead of the USD.10$ fee, we Holo would earn USD$0.01. These transactions are in the gray area where existing Cryptocurrencies have similar fees on this scale of transactions; this cap would establish Holo fuel as a clear winner against existing Cryptocurrencies, and every Fiat currency payment processing system.

Transactions between ~USD$10-$10,000.00 are the bulk of normal consumer retail transactions, and the the most price sensitive. Most importantly, these are the personal short-term savings "stock" and day-to-day retail "flow" of money; by far the largest pool of savings and monetary velocity.

A fee of 1% on these purchases (or account maintenance activities) would be considered onerous vs. virtually any other payment system. Presently, retailers do bear significant losses due to fees on these transactions in Fiat currencies – but this is largely hidden from consumers by contractually enforced (and craftily mandated) bundling of these fees into overall retail sales overheads.

This is also the range where existing cryptocurrencies excel; the fees are much lower than for Fiat systems. For Holo fuel, consumers would not be happy paying a USD$10 direct fee (1%) on a USD$1,000 bicycle, or an extra USD$100 on a used car purchase, and it materially would affect usage of Holo fuel for general retail purchases of all sizes. By limiting the fees at 0.1% on USD$10-$1,000 (fees of USD$0.01-$1.00), and capping USD$1,000-$10,000 to a max. USD$1.00 fee, the fee ceases to be a negative psychological factor in these purchases; Holo fuel will be the lowest cost option available for retail level savings and transactions.

The losses due to this fee/cap are likely to be negligible, since Holo fuel would not have been competitive for these transactions anyway, and would not likely have been used. Now, Holo gains additional revenue that it would not have otherwise earned, even though the rate is lower than for micro/small transactions.

For large transactions, once again cryptocurrency excels. However, businesses would likely find a 0.1% fee on **all** their earnings significant; A business that earns USD$1,000,000/yr would bear an additional USD$1,000 expense due to accepting Holo fuel. This would be noticeable to accountants and business owners, and would affect uptake negatively. By limiting this to .01% and capping fees at USD$1,000, the fees on USD$10,000-10,000,000 would be USD$1-1,000 on large transactions; roughly the cost in annual fees for a business bank account. This

would ameliorate any objections to Holo fuel uptake at the boardroom level. Once again, a 1% or 0.1% fee would be a non-starter, so any losses in fee income to Holo are imaginary.

Finally, since Holo fuel and the Holo Reserve ecosystem need to be able to identify accounts belonging to Holo Host DPKI identities in order to authorize access to Reserves, the infrastructure must be built to identify that accounts are "related" to eachother under the same identity. Paying fees on transfers between accounts should be eliminated, in order to remove any psychological barriers to uptake resulting from seemingly unnecessary fees being charged just to organize money between personal or business accounts. Freeing up the Holo fuel users to feel good about organizing their funds is a critical determinant of uptake, and any perceived Holo fee losses once again are illusory; the money is going to eventually be spent, and would (otherwise) just sit uncomfortably in fewer accounts.

In summary: 1% fees are fine, so long as caps and lower fee are established and larger transactions. Holo can't earn **any** fees on transactions that never occur; and alienating all users of retail, business and large-scale transactions is a recipe for failure; Holo fuel will forever be relegated to microtransactions in the strictly limited economy of Holo, instead of taking its rightful place as a linchpin currency of trade and commerce.

4. Units circulating

   Currency units available to buy/sell in trades define the value of the currency, in terms of external benchmarks. This is the sum of all units created by all credit lines (including the initial 177e9 HOT issued by Holo, plus all Reserves and other credit lines), minus the units kept in savings (positive balances and un-deployed credit lines). Units of Holo fuel are circulated in trade for goods and services, on various Exchanges, etc.. Some holders of these units will be seeking to trade them on various Reserves.

   If there are 1M units of currency available on the market at an "ask" price of USD$1, and ~USD$1M of trade into the currency to execute, exactly 1M of units will be purchased by potential trade buyers, exchanged for the USD$1M of goods; the unit value of the currency is in equilibrium at an exchange rate of USD$1/unit. On the other hand, if there are 1010101 units of currency available at an "ask" of USD$0.99, and 1 unit available at an "ask" of USD$1.00, then the trade buyer would purchase the 1010101 units to satisfy their USD$1M trade requirements, leaving the current "ask" price at USD$1.00; the currency is now in equilibrium at an exchange rate of USD$1.00/unit.

5. Units Issued by Credit

   Until the value of Holo fuel nears its value-stability target, it makes no financial sense for anyone to issue or use credit. Unless you believe you can **deploy** the credit at a rate of return greater than the expected future deflation rate of the currency, you are going to lose money on the investment. This is the inverse of the card Fiat currency holders are usually dealt: under inflation, it makes no sense to **save** in units of the currency, unless you know you can deploy the currency (eg. deposit at interest) at a rate of return the exceeds the inflation rate.

   So, until that point, while Holo fuel is floating and generally deflationary (trade in the currency is increasing over time), neither Holo (via Reserve issuance) nor the Hosts (via Credit lines) should issue Holo fuel. Once the market value reaches the plateau where Hosts are willing to redeem their earned Holo fuel for various Fiat currencies, Holo Reserves can begin to sell Holo fuel at that price for Fiat, and simultaneously redeem Holo fuel for that Fiat currency, on a LIFO basis. If Holo gets stuck with Fiat currency in the Reserve at a rate Hosts are no longer willing to accept, it may form a blockage to older an more desirably priced reserves;

Holo also has the ability to access reserves in LIFO order, and redeem its own Transaction Fee Holo fuel income for these less desirably priced blocks of Fiat currency reserves.

When target unit valuation is achieved, **so long as trade in the currency is still generally expanding**, Holo reserves can establish upper/lower bounds; for example (if the currency was pegged to USD$1), the Reserves could place a standing "ask" (sell) order at USD$1.01 per unit, and a "bid" at USD$0.99 per unit ('til its reserves were exhausted). For every 1M units of currency issued at USD$1.01 during a deflationary period (eg. all circulating units sold, remaining units in savings), 1020202 units could be redeemed at USD$0.99, removing 20202 units from circulation during a subsequent period of inflation (eg. a HODLer decides to liquidate).

The PID loop controlling the credit factor extended to Hosts on the NPV of their hosting services (or other wealth holders) would be tuned to actively control credit factor K between USD$1.01 and USD$0.99, increasing and decreasing the scale of the credit lines extended on the value of the attached wealth. Some of these credit line holders would decide to establish "ask" (sell) orders for some of their credit line at say USD$1.009 and a "bid" at USD$0.991 (to beat the order place by the Holo Reserves). They would have the authority to issue units of currency up to their credit limit during deflation (high currency prices). In aggregate, all issuers would reflate the money supply; the PID loop would detect the flattening rate of change (Derivative term), pulling back on the credit factor K as the currency value nears its target value. Later, during inflation (low currency prices), they would purchase the units back at a discount.

6. Units in Savings

   Units of currency not in circulation are held in savings, until the owners are presented with opportunities that are more compelling than holding the currency. During deflation (expanding trade in the currency), units will be sold on a need basis, or when the risk of loss compels diversification. After value stability, savings in a value-stable currency provide a safe haven for funds while seeking investment. All of these scenarios remove units of currency from circulation, demanding issuance/withdrawal of credit to balance supply/demand for equilibrium valuation to be maintained.

7. Multiple Degrees of Freedom on Holo fuel Valuation

   Thus, there are 2 degrees of freedom that holders of Holo fuel need to mentally deal with, which are largely independent of eachother, and are influenced by separate factors.

   - The value of a unit of Holo fuel in compute resources (hours of computer time)
     - How much Holo fuel should I sell my Holo hosting services for?
       - Defined by the market value of the underlying Holo computation system
   - The price of a unit of Holo fuel in various Fiat currencies (eg. exchange rate to USD$)
     - What will I earn when I sell that fuel on the Reserves or other exchanges?
       - Defined by what the market believes Holo fuel is "worth"; Utility +

   Fixing the value-stability of Holo fuel is **independent** of the amount of compute resources the Host eventually redeems in Fiat currency; it doesn't matter if 1 month of compute resources is sold for 100 Holo fuel at a USD$1.00/Holofuel, or 10,000 Holo fuel at USD$0.01/Holofuel – the Host earns USD$100 per month of compute. The Holo fuel valuation in terms of compute

resources could be maintained at an arbitrary value by limiting the inflow of Holo fuel into the core Holo ecosystem, without affecting the freedom of movement of its exchange rate with Fiat currencies.

Removing this one degree of freedom from the Holo fuel system does **not** restrict the market's ability to price Holo hosting! The market price (how many USD$ buys 1 Holo-month of hosting) is arrived at by the market reaching equilibrium. What everyone **does** now, however, is how much 1 Holo represents in terms of Holo hosting computational resources. This allows Holo fuel to act both as a medium of exchange (which does not require value stability), **and** as a value reference (which **does** require stability in terms of some reference commodity basket.) Both are required for Holo fuel to be used as general "money". Uncontrolled large short-term swings in valuation will absolutely invalidate it for short-term retail savings/spending; the public cannot risk their day-to-day money's value, and businesses cannot bear the cost of re-pricing their inventory on a daily (or hourly) basis.

Initially, Holo has guaranteed convertability 1-1 between HOT and Holo fuel. Thus, there will be a large influx of HOT into the Holo system. This will result in a potentially very large amount of liquid Holo fuel available to Hosts and Holo hApp owners, mapped onto a quite small pool of Host compute resources, if the primary utilization of Holo fuel is just buying and selling Holo compute. However, if the HOT is not converted, and/or the Holo fuel created upon deposit of the HOT in the HOT/Holofuel Reserve account is used for arbitrary economic activity (savings, exchange for general goods and services), then the amount of Holo fuel used within the Holo ecosystem in exchange for Holo hosting may remain low, allowing us to begin issuing/withdrawing Holo fuel credit lines to control inflation/deflation.

Thus, the size of the basket of compute resources that Holo fuel eventually settles on depends on the amount available for Holo hApp owners to purchase to buy Holo hosting. If all HOT$177e9 is available (nobody uses/saves it for anything else), and it starts at the current USD$0.001/Holo fuel, the total in circulation is worth USD$177,000,000. To get 12 circulations of that Holo fuel per hear (Holo hApps and Hosts only saving about 1 month's worth), the compute infrastructure would have to be worth about USD$2.2 billion / year in sales – about 1/6th the scale of Amazon AWS (which earned USD$3.3 billion in 2017 on USD$12.2 billion in sales). Since Holo can be deployed on commodity hardware, but doesn't support commodity disk images (only Holochain applications), it is difficult to imagine that scale of compute happening in the short term.

This is not the most likely scenario. Holo fuel has **significant** benefits vis a vis existing cryptocurrencies. These currencies have significant complexity, risk and scalability issues that limit uptake, which are solved by Holo fuel's Holochain underpinnings, and yet achieve high valuations based on utility alone. **If** Holo addresses the high fee issues (eg. capped the 1% fee on all Spender transactions at the equivalent of USD$0.01, or 1 cent), then the uptake for non-Holo transactions could be essentially **unlimited**. This would support the influx of wealth from other cryptocurrencies and Fiat deposits which could quickly approach USD$177,000,000,000 (177 billion) in value. The total market capitization of all cryptocurrencies is estimated at about USD$224 billion as of September 2018; roughly the value of the residential real estate in 1 small city.

Quenching deflation (decreasing commodity prices, increasing per-unit value) at a certain target unit value is simple, as long as enough people have attached enough wealth to the system: begin issuing Holo fuel credit against wealth (eg. the net present value of a Holo hosting service). Even if that target value is low (say, USD$0.01 or USD$1.77 billion, and if

the Holo system generated USD$5.00 per core per month in revenue, the NPV of one year's worth of 1 core worth (USD$60) of hosting at a 10% discount rate is USD$54.55. Estimating a credit factor of 50%, that results in USD$27.27 of credit extended per core. Extending approximately the same amount of credit as the valuation of the already issued tokens would require about 65 million cores of Holo Hosting power online; not immediately practical.

But, value-stability in Holo fuel at a significant per-unit valuation requires much more than that. The ability to also quench inflation (increasing commodity prices, lowering per-unit values) in Holo fuel requires that we be able to withdraw a large fraction of the circulating supply, by reducing credit lines. If the existing Holo fuel credit created via issuance of credit against wealth is small vs. the total supply (as it could be at the start of the value-stable period with Holo fuel), and there was a significant reduction in Holo fuel financial activity or a large HODLer began dumping their savings, at least that amount of Holo fuel credit would need to be withdrawn. To avoid significant reductions in each credit account, the total amount issued in credit has to be very large vs. the amount of Holo fuel that needs to be withdrawn. So, total credit issued needs to be several multiples of the amount in circulation **not** created through credit lines (eg. that originating from the originally minted HOT ERC20 tokens).

The only practical way to achieve that is to allow **other** forms of wealth to be attached and monetized into Holo fuel credit; for example, houses, vehicles, monetary savings and non-monetary wealth owned unencumbered by liens or other restrictions.

2.2.3 Modelling Holo Fuel

In these models, our goal is to observe how Holo fuel Reserves respond to variations in Fiat currency relative valuations, and to variations in how existing HOT ERC20 tokens are converted to Holo fuel, and processed by varying scales of Holo Hosting and varying levels of "retail" savings and transaction levels.

We will not be controlling Holo fuel's value stability in this model; see Holo Fuel Currency System Design and Modelling to see how value-stability in terms of a basket of commodities can be implemented. Holo fuel's value will float, both in terms of its value reference "basket" of Holo computational commodities, and its exchange rate in Fiat currency.

When prices rise (Client requests increase, Host load increases prices), eventually they will reach the Reserve agent sellers' ask price, derived from the Hosts' cash-out settings. This will result in the issuance of more Holo fuel, moderating price deflation. Basically, Holo fuel prices should cap out at Host cash-out price. We can limit Reserve issuance and/or increase prices to allow further increase in Holo fuel price, but we won't do that in this model.

Slower hosts are priced cheaply, faster hosts are more expensive. This is an aggregate of all types of requests made to a hosted hApp, so represents the full spectrum of Host behaviours (requests that are not satisfiable in a deterministic time should be excluded for the purposes of Host characterization). For example, a very fast host on a low-latency network but with slow disk storage will be penalized vs. an identical host with SSDs, because its disk-intensive requests will have a response time distribution with higher mean and standard deviation. However, it may offset this by pricing its 'storage' and 'bandwidth' commodities at a premium. For the purposes of the model, various Hosts will satisfy requests at various rates, but they will all be considered to be in one pool. Normally, a host that takes longer to process requests would migrate to a lower performance pool, where its price would probably move it to the **premium** price tranche in that performance band, reducing its request rate, and hence lowering its income.

Hosts may chose to sell fuel on the Reserve (which is also an exchange in this simulation) to

whomever is buying at market rates, to maintain their monthly cost needs, or may hold onto fuel until they can sell at their cash-out price, either to the Reserve agent (Hosts can access Fiat reserves in LIFO order), or to any other buyer. Holo hApps will buy on the Reserve/exchange to maintain service, either from private sellers, or from the Reserve agent at the Reserve ask price.

As prices increase HODLers should begin liquidating some of their HOT$177e9 in Holo fuel holdings to furnish this need. They would sell on a HOT/fuel exchange to do this, but we'll just simulate everyone holding Holo fuel buying and selling on the Reserve exchange; however, only Hosts are authorized to sell (redeem) Holo fuel to the Reserve agent at LIFO rates.

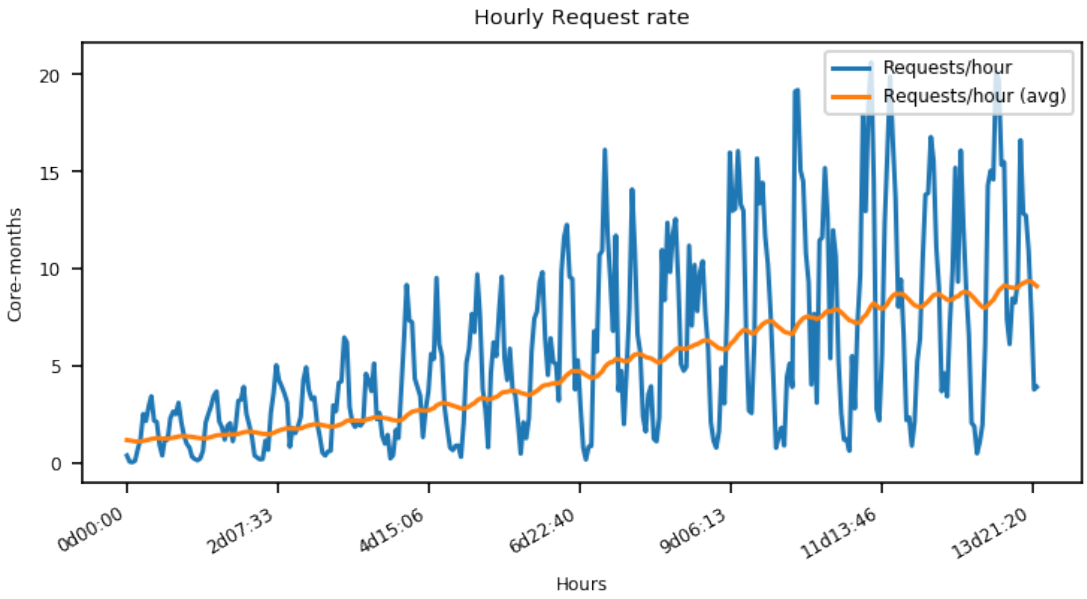
```
def std_dst_prob( SD ):
    """Given a number of SD away from the mean, compute the probability of that number being part of the
    normal distribution.  For example, if we're +2 standard deviations away, we're in the 97.7th
    percentile; only 2.23% of the population should exceed this value in a normal distribution.  At
    0 SD away, we're right on the 50th percentile; 1/2 should be less, 1/2 more.  We want a function
    that, given a SD, provides us a probability of 1.0 at exactly 0 SD away, and falls off in the
    shape of the Bell Curve as we retreat from the mean; At 0.0, we want 2 x 0.5 == 1.0; At +2.0 or
    -2.0 SD away, we want the result == 2 * 0.0227 == 0.0454 .

        >>> stats.norm.cdf( 0 )
        0.5
        >>> stats.norm.cdf( +2 )
        0.9772498680518208
        >>> stats.norm.cdf( -2 )
        0.022750131948179195
        >>> 1 - stats.norm.cdf( +2 )
        0.02275013194817921

    """
    if SD < 0:
        return 2.0 * scipy.stats.norm.cdf( SD )
    else:
        return 2.0 * ( 1 - scipy.stats.norm.cdf( SD ))


def exponential_moving_average( current, sample, weight ):
    return sample if current is None else current + weight * ( sample - current )


class Client_hApp( trading.actor ):
    """A client tries to perform a certain number of requests per hour (via an hApp), during a 12-hour
    window of time peaking around some time during the day.  Our quanta is 1 hour, so we'll
    recompute our next hour's requests every hour -- as well, each Client will have a random start
    time during the first hour.  So, if we sample all the client's self.requests at the Nyquist
    rate, we'll have a good instantaneous view of the current request rate, in request/hr.

    Each Client_hApp represents a block of actual clients; the requests_mean is denominated in 1 x
    'holo' worth of requests; a month of Host core wall-clock duration of requests to service some
    number of individual active users (Clients); this varies based on the computational complexity
    of the hApp (per-request wall-clock avg. duration) and the request rate (how many requests
    required per client visit) to sustain each client.

    So, this doesn't represent a certain number of clients; it represents a certain hosting load
    required to satisfy a pool of clients in some geographical locale with a certain request rate
    pattern, peaking at the target request_mean (eg. 10 holo, or 10 wall-clock months of Hosting
    load) during each quanta (eg. 1 hour).  So, for a typical small hApp which might be able to
    sustain 10 requests/second (avg. 100ms per request at 100% load, or 300ms per request at 33%
    load eg. due to blocking I/O; remember: Holo hosts distributed apps, so most requests will
    perform network I/O; static content, and simple programmatically generated content will be
    served by CDNs or simple webservers, not Holo).  If it uses 10 'holo' (Host-months) worth of
    hosting in an hour, then the Holo Hosting system is serving 10 * 2,629,800 seconds, or around
    26,298,000 requests during that hour.  If the average online client consumes 100 request to
    execute their activity during that hour, we are representing the activity of 262,980 unique hApp
    users.
```

```
"""

# Increase the reqwuest_clients for each Client_hApp by this multiplier
requests_clients_multiplier = 1

def __init__( self,
        midday_mean = 12 * trading.hour, midday_sigma = 2,  # noon, +/- 2 hours has 2/3 of requests
        requests_mean = 1.0, requests_sigma = .2,           # 1 'holo' worth of req/h, +/- .2 peak
        requests_clients = 2500,                            # This represents this many clients (lower == heavier hApp)
        quanta = 1 * trading.hour,                          # compute next hour's requests hourly
        requests_avg_dur = 2 * trading.day,                 # keep ?-day rolling average of request rate
        holofuel_runway = 1 * trading.day,                  # target this many day's fuel on hand
        minimum = -math.inf, **kwds ):                      # allow client to go into debt
    # dummy (zero) 'needs'; adjust targets manually, but ensures that Holofuel target satsifying
    # trades are executed.
    super( Client_hApp, self ).__init__(
            quanta          = quanta,
            minimum         = minimum,
            needs           = [ trading.need_t( 0, None, 'Holofuel', quanta, 0 )], **kwds )
    self.midday_mean        = midday_mean                   # eg. -7 (Mountain), +5 (China Standard)
    self.midday_sigma       = midday_sigma
    self.requests_mean      = requests_mean
    self.requests_sigma     = requests_sigma
    self.requests_clients   = requests_clients
    self.requests           = 0
    self.unsatisfied        = 0
    # avg. should be somewhere around here
    self.requests_avg       = requests_mean * ( 3 * midday_sigma/2 / 24 ) # 99.8% w/in 3 sigma of mean

    self.requests_avg_dur   = requests_avg_dur
    self.compute_requests_rate( now=self.start )

    self.holofuel_runway    = holofuel_runway

#
# .clients       == the number of Client sessions
# .requests      == 'holo' load represented by all .client's requests during .dt
# .requests_avg  == exponential moving average of load
#
@property
def clients( self ):
    return self.requests_clients * self.requests_clients_multiplier

def compute_requests_rate( self, now ):
    """Time quanta reached; .now updated, .dt has time period since last run.  Compute our next hour's
    number of satisfied/unsatisfied requests, based on the last hour's Holo system thruput.
    Thus, when Clients want to perform 10 tx but the last hour saw 125% utilization of Host
    resources, the next hour we'll compute the target rate we'd like (say, 10 requests), but
    reduce it by / 1.25, and say 8.0 satisfied, 2.0 unsatisfied.  """
    peak_desired            = rnd_std_dst( mean=self.requests_mean, sigma=self.requests_sigma, minimum=0 )
    peak_desired           *= self.requests_clients_multiplier

    # how many hours -/+ from our midday peak utilization?  Pick a random hour we want to peak,
    # somewhere near our desired "midday".  Lets pick a random normal value around our target,
    # and then compute our Z-score: what is the probability of something being in the normal
    # distribution, that far from the mean.  For example, if our curr_hour is 11:00, and our peak
    # hour comes out to be exactly 12:00, we're -1 hour away.  If our sigma (size of 1 standard deviation)
    # is 6 hours, we're -1/6th SD away from the mean.
    curr_hour               = ( now % trading.day ) / trading.hour                          # (0,24] UTC
    peak_hour               = rnd_std_dst( mean=self.midday_mean, sigma=self.midday_sigma ) # (-12,+12)
    norm_hour               = ( peak_hour + 24 ) % 24 # convert eg. timezone -7 to +17 hour of UTC day
    diff_hour               = curr_hour - norm_hour
    if diff_hour < -12:
        diff_hour          += 24
    elif diff_hour > +12:
        diff_hour          -= 24
    #print( "midday_mean: {self.midday_mean}, peak: {peak_hour}, curr: {curr_hour}, norm: {norm_hour}, diff: {diff_hour}".fo
```

```
        curr_sd                 = diff_hour / self.midday_sigma
        curr_hour_prob          = std_dst_prob( curr_sd )
        hour_target             = peak_desired * curr_hour_prob
        #print( "curr/peak hour: {} vs. {}, curr_sd: {}, hour_target: {}, prob: {} ".format(
        #    curr_hour, peak_hour, curr_sd, hour_target, curr_hour_prob ))
        self.requests           = hour_target
        self.unsatisfied        = 0 # TODO: get this from Host average utilization
        self.requests_avg       = exponential_moving_average( self.requests_avg, self.requests, self.quanta / self.requests_avg_

    def run( self, **kwds ):
        """Runs the Client/hApp trading.actor's needs/targets and issues trades as required to get
        self.targets satisfied.
        """
        if not super( Client_hApp, self ).run( **kwds ):
            return False # Quanta not yet satisfied; do nothing

        # Transfer our last hour's request rate Hosting payment in Holofuel, to a random Host.
        host                    = random.choice( hosts )
        cost                    = self.requests
        try:             host.assets['Holofuel'] += +cost
        except KeyError: host.assets['Holofuel']  = +cost
        try:             self.assets['Holofuel'] += -cost
        except KeyError: self.assets['Holofuel']  = -cost
        logging.info( "Transferring {} Holofuel from {} to {}: {}".format( cost, str( self ), str( host ), repr( host.assets )))
        # Compute our next hour's Request rate, based on time of day
        self.compute_requests_rate( now=self.now )

        # We've computed our requests_avg Requests/hour.  W'll adjust self.targets manually to keep
        # (say) a week's worth of runway Holo fuel on hand.  We'll be paying Hosting fees hourly for
        # the last hour's requests, out of our Holo fuel assets, reducing them, triggering a buy to
        # bring us back up to targets.
        self.target['Holofuel'] = self.holofuel_runway * self.requests_avg / self.quanta
        return True


class Sample_engine( trading.engine_status ):
    """A trading.engine that takes a sample of the world's constituent agents every status_period.
    These samples are used to produce the graphs.
    """
    def __init__( self, **kwds ):
        super( Sample_engine, self ).__init__( **kwds )
        self.requests           = [] # [ (<now>, <'holo'-capacity per quanta>,<client-requests>,<client-request-avg>), ... ]
        self.clients            = [] # [ (<now>, <#clients>), ... ]
        self.cores              = [] # [ (<now>, <#cores>), ... ]
        self.hApp_holdings      = [] # [ (<now>,{'Holofuel': 123, 'USD':-123}), ... ]
        self.Host_holdings      = []

    @property
    def hApps( self ):
        for a in self.agents:
            if isinstance( a, Client_hApp ):
                yield a

    @property
    def Hosts( self ):
        for a in self.agents:
            if isinstance( a, Host ):
                yield a

    def status( self, now ):
        """Collect hourly snapshots of all of our Client/hApps' simulated requests for that hour and rolling avg."""
        super( Sample_engine, self ).status( now=now )
        self.requests.append( (now, sum( h.capacity for h in self.Hosts ),
                                    sum( c.requests for c in self.hApps ),
                                    sum( c.requests_avg for c in self.hApps )) )
        self.clients.append(  (now, sum( c.clients  for c in self.hApps )) )
        self.cores.append(    (now, sum( h.cores    for h in self.Hosts )) )
        self.hApp_holdings.append( (now, {
```

```
            'USD':      sum( c.balances.get( 'USD' ) or 0 for c in self.hApps ),
            'Holofuel': sum( c.assets.get( 'Holofuel' ) or 0 for c in self.hApps )
        }) )
        self.Host_holdings.append( (now, {
            'USD':      sum( c.balances.get( 'USD' ) or 0 for c in self.Hosts ),
            'Holofuel': sum( c.assets.get( 'Holofuel' ) or 0 for c in self.Hosts )
        }) )
        logging.info( "{}: Holofuel Target: {}, Holdings: {}, USD: {} Order Book:\n{}".format(
            str( self.world ),
            sum( c.target.get( 'Holofuel' ) or 0 for c in self.hApps ),
            sum( c.assets.get( 'Holofuel' ) or 0 for c in self.hApps ),
            sum( c.balances.get( 'USD' ) or 0 for c in self.hApps ),
            self.exchange.format_book() ))

class Sample_engine_inc_clients( Sample_engine ):
    def status( self, now ):
        try:
            return super( Sample_engine_inc_clients, self ).status( now )
        finally:
            # Adjust the simulation on each status cycle.  Increase client load by 1.25% / period,
            # declining over the duration of the simulation
            Client_hApp.requests_clients_multiplier *= ( 1.0 + .0125 * ( 1 - now / sim_duration ))

#
# Create varying populations with different request activity times
#
pop_t                           = collections.namedtuple( 'Population', ['locale', 'tz', 'P'] )
populations                     = [
    pop_t( 'NA-west', -8,  100e6 * .9 ),
    pop_t( 'NA-mid',  -6,   75e6 * .8 ),
    pop_t( 'NA-east', -5,  150e6 * .9 ),
    pop_t( 'Africa',  +2, 1.21e9 * .2 ),
    pop_t( 'EU',      +0,  750e6 * .7 ),
    pop_t( 'RU',      +3,  150e6 * .6 ),
    pop_t( 'Asia',    +8, 1.38e9 * .3 ),
]

def population_middays( count, *matches ): # eg. 'NA', 'Asia'; empty matches == everything
    pops_selected               = [ p for p in populations for m in ( matches or [''] ) if m in p.locale ]
    pop_prob                    = [ p.P for p in pops_selected ] # population * middle-class,technical
    pop_prob                   /= np.sum( pop_prob ) # normalize sum of probabilites to 1.0
    pop_tz                      = [ p.tz for p in pops_selected ]
    pop_middays                 = np.random.choice( a=pop_tz, size=count, replace=True, p=pop_prob )
    #print( ', '.join( "{:>10}: UTC {:+d} P({:.2f}):".format( locale, tz, prob )
    #                  for locale,prob,tz in zip( (p.locale for p in pops_selected ), pop_prob, pop_tz )))
    return pop_middays


#
# Render a duration of client_count Client's request activity.  Each Client_hApp represents a set of
# hApp client groups w/ a certain average midday peak load and standard distribution.  Create
# clients that request a certain standard distribution of Holo Hosting load per hour.  Each of the
# clients will represent a pool of users in that time zone, w/ some mean request rate (denominated
# in 'holo': Hosting-months of wall-clock request duration), in a standard distribution around some
# peak midday mean time.
#
client_count                    = 25

# Peak session load at "midday" hour; compute 'holo' (Holo hosting-core-months) used to satisfy requests
# 1000 * 100 * 10 / 2.62e6 =~= 0.382 'holo' per peak hour
requests_clients                = 1000 # How many client sessions does each Client_hApp simulate?
requests_per_session            = 100 # eg. a browsing Facebook, booking a vacation
ms_per_request                  = 10 # wall-clock duration probably higher, due to blocking I/O
requests_mean                   = requests_clients * requests_per_session * ms_per_request / trading.month
requests_sigma                  = requests_mean * 0.2 # +/- 20% w/in 1 sigma

clients                         = [ Client_hApp(
                                        identity        = "hApp {}".format( n ),
```

```
                                        midday_mean     = midday_mean,
                                        requests_mean   = requests_mean,
                                        requests_sigma  = requests_sigma,
                                        requests_clients= requests_clients )
                                    for n,midday_mean in enumerate( population_middays( client_count, 'NA', 'Asia' )) ]


class reserve_redeeming_Hosts( reserve_issuing ):
    """Only allow our market-maker to buy (redemm) Holo fuel from Hosts.  We'll sell Holo fuel to anyone,
    and other actors may trade with eachother, but our own tranches are unavailable to anyone but Hosts.

    TODO: Interogate Hosts, and ensure that they cannot redeem more Holo fuel than they have earned
    due to Hosting.
    """
    def __init__( self, *args, **kwds ):
        super( reserve_redeeming_Hosts, self ).__init__( *args, **kwds )

    def buys_from( self, another ):
        allowed = isinstance( another, Host ) and super( reserve_redeeming_Hosts, self ).buys_from( another )
        logging.info( "%s buys from %s: %s", self, another, allowed )
        return allowed


res                             = reserve_redeeming_Hosts( "Holofuel/USD", LIFO=True, supply_period=trading.hour, supply_availab
wld                             = trading.world( duration=sim_duration, quanta=trading.hour )
eng                             = Sample_engine_inc_clients(
                                      world=wld, exch=res, agents=clients + hosts, status_period=1 * trading.hour / 2 ) # Nyquis
#print( str( eng.world ))
eng.run()
#print( "Done: %s" % ( eng.world ))


x_now,y_cap,y_req,y_avg         = [],[],[],[]
if eng.requests:
    x_now,y_cap,y_req,y_avg     = zip( *eng.requests )

fig,ax = plt.subplots()
plt.plot( x_now, y_req, label="Requests/hour" )
plt.plot( x_now, y_avg, label="Requests/hour (avg)" )
formatter = matplotlib.ticker.FuncFormatter( lambda s, x: '%dd%02d:%02d' % (
    s // trading.day, ( s % trading.day ) // trading.hour, ( s % trading.hour ) // trading.minute ))
ax.xaxis.set_major_formatter( formatter )
fig.autofmt_xdate()
plt.xlabel( "Hours" )
plt.ylabel( "Core-months" )
plt.legend( loc="upper right" )
plt.title( "Hourly Request rate" )
plt.show()
```



So, here we observe the Client/hApp hourly request load generated by a number of clients in a couple of time zones, over a few days; Asia is bigger and more concentrated, NA is more spread out. Of course, we'll spread out the midday times to simulate loads coming from various sizes of populations.

The request rate is load/capacity is denominated in 'holo'; 1 'holo' is 1 core-month of requests served by a Host. This capacity is generally linearly related to the number of cores on the Host, but is also affected the CPU speed, RAM speed, bandwidth and latency and storage speed. We'll assume our Hosts are pretty homogeneous.

We'll increase the number of simulated Clients over the period of the test.

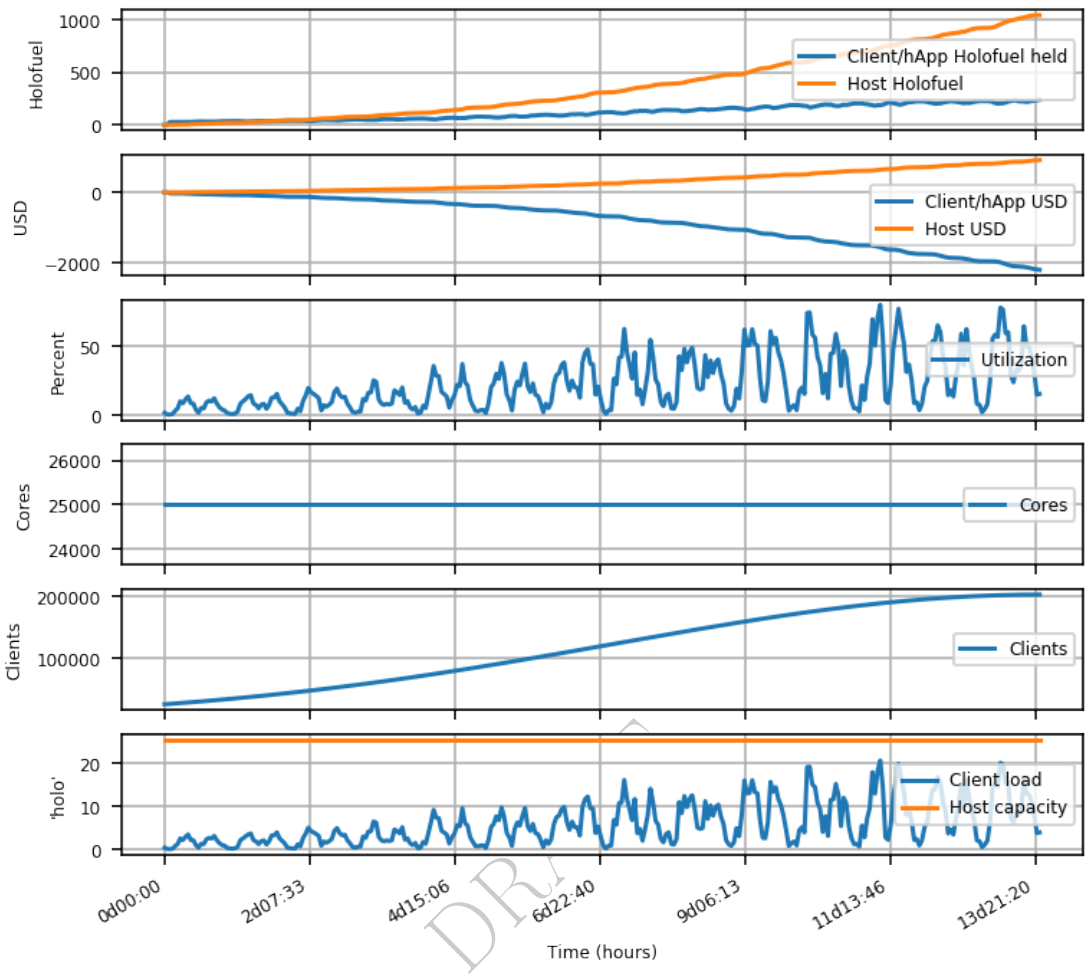
Instead of simulating the hApp owner buying Holo fuel and paying it to the Host, we'll simply have the Clients directly buy the Holo fuel required to service their requests. Like a hApp owner, they'll need to buy it at "market" for that period's worth of requests.

The "Requests/hour (avg)" gives the Client/hApp the information it needs to fund future requests. We ensure that our Client/hApps have always got enough Holo fuel to fund a week's worth of hosting.

Holo fuel is transferred to a random Host on an hourly basis by each Client/hApp. Here we see the purchases of Holofuel from the Reserve being transferred to the Hosts, and the increasing total USD$ cost of Hosting accruing to the Client/hApps:

```
fig,(ax0,ax1,ax2,ax3,ax4,ax5)= plt.subplots( 6, sharex=True, figsize=(6,6) )

ax0.plot( [ x for x,a in eng.hApp_holdings ], [a['Holofuel'] for x,a in eng.hApp_holdings ],
              label='Client/hApp Holofuel held' )
ax0.plot( [ x for x,h in eng.Host_holdings ], [h['Holofuel'] for x,h in eng.Host_holdings ],
              label='Host Holofuel' )
ax0.fmt_ydata = lambda x: '%.2f' % x
ax0.grid( True )
ax0.set_ylabel( 'Holofuel' )

ax1.plot( [ x for x,h in eng.hApp_holdings ], [h['USD'] for x,h in eng.hApp_holdings ],
              label='Client/hApp USD' )
ax1.plot( [ x for x,h in eng.Host_holdings ], [h['USD'] for x,h in eng.Host_holdings ],
              label='Host USD' )
```

```
ax1.fmt_ydata = lambda x: '%.2f' % x
ax1.grid( True )
ax1.set_ylabel( 'USD' )

ax2.plot( [ x for x,cap,req,rav in eng.requests ], [ req/cap * 100 if cap else 0 for x,cap,req,rav in eng.requests ],
              label='Utilization' )
ax2.fmt_ydata = lambda x: '%.0f%%' % x
ax2.grid( True )
ax2.set_ylabel( 'Percent' )

ax3.plot( [ x for x,cores in eng.cores ], [ cores for x,cores in eng.cores ],
              label='Cores' )
ax3.fmt_ydata = lambda x: '%.0f' % x
ax3.grid( True )
ax3.set_ylabel( 'Cores' )

ax4.plot( [ x for x,cli in eng.clients ], [ cli for x,cli in eng.clients ],
              label='Clients' )
ax4.fmt_ydata = lambda x: '%.0f' % x
ax4.grid( True )
ax4.set_ylabel( 'Clients' )

ax5.plot( [ x for x,cap,req,rav in eng.requests ], [ req for x,cap,req,rav in eng.requests ],
              label='Client load' )
ax5.plot( [ x for x,cap,req,rav in eng.requests ], [ cap for x,cap,req,rav in eng.requests ],
              label='Host capacity' )
ax5.fmt_ydata = lambda x: '%.0f' % x
ax5.grid( True )
ax5.set_ylabel( "'holo'" )

ax5.set_xlabel( "Time (hours)" )
formatter = matplotlib.ticker.FuncFormatter( lambda s, x: '%dd%02d:%02d' % (
    s // trading.day, ( s % trading.day ) // trading.hour, ( s % trading.hour ) // trading.minute ))
ax4.xaxis.set_major_formatter( formatter )
fig.autofmt_xdate()

for a in ax0,ax1,ax2,ax3,ax4,ax5:
    a.legend( loc="right" )
```



This illustrates the creation of Holo fuel via Reserves at a fixed issuance rate of USD$1/Holofuel. No other sellers are initially available, so Client/hApp buyers are forced to purchase at the Reserve's issuance rate. Once Hosts have Holo fuel, they begin to cash out at their desired exchange rate.

The Host's charged prices in Holo fuel are fixed, and the hApp owners are willing and able to pay whatever is demanded. Of course, none of these assumptions reflect a real marketplace.