

SagaPython Walkthrough

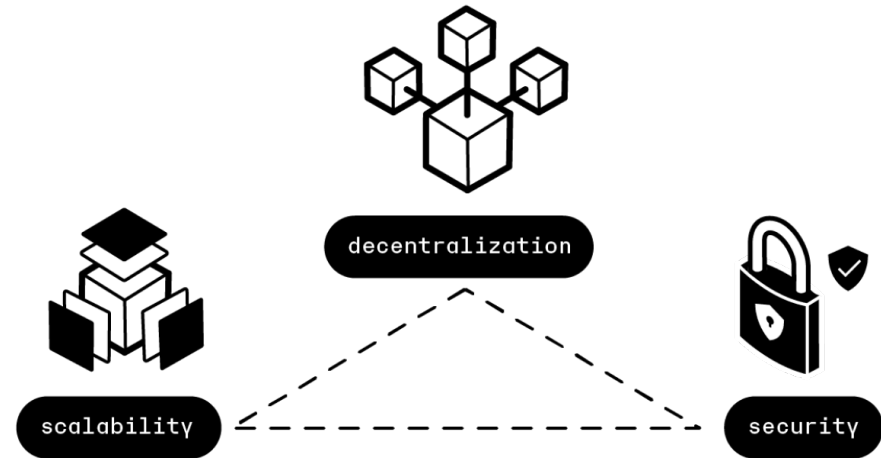
PraSagaTM

ReImagine. ReThink. ReInvent.TM

The background of the slide is a solid yellow color. It features a complex, abstract pattern of overlapping geometric shapes, including large circles and squares, in various shades of yellow and gold. The shapes are arranged in a way that creates a sense of depth and movement. The text "SagaChain Motivation" is centered in the middle of the slide, written in a white, sans-serif font. The text has a slight drop shadow, making it stand out against the busy background.

SagaChain Motivation

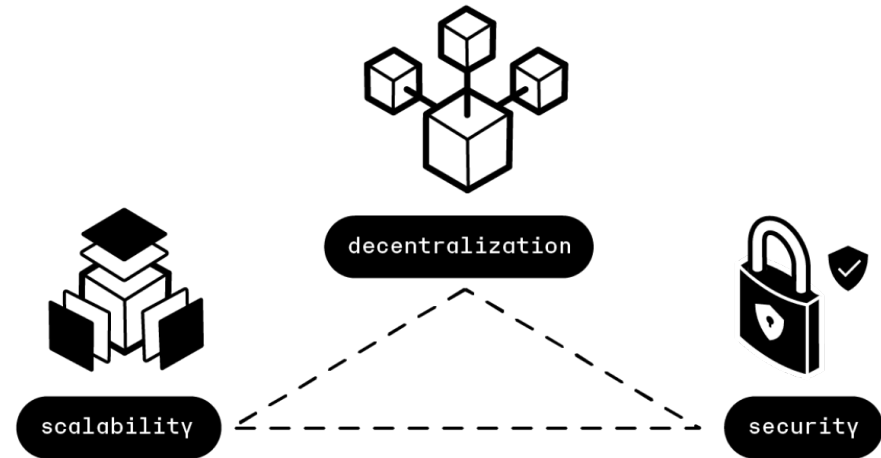
Rethinking the Trilemma



- Define Scalability as capacity of transactions per unit time with a constant latency per transaction.
- Scale is equated with parallelization (i.e. execution sharding)
- Decentralization held constant with increasing capacity
- Security held constant with increasing capacity

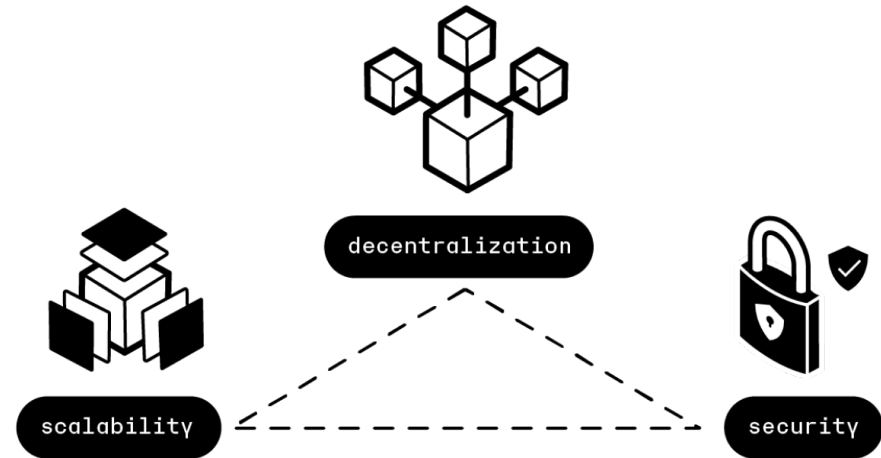
Design objective: increase parallelization (i.e. capacity) with increasing number of validator nodes

Rethinking the Trilemma



- Parallelization requires a solution to data distributed across multiple shards
- Must guarantee deterministic execution; implies synchronization of data across multiple shards to maintain state replication.
- Thus core problem to solve is synchronized state replication across multiple shards, such that transactions can safely be executed in parallel without introducing nondeterminism

Rethinking the Trilemma



- Ideal situation:
- All shards can execute transactions in parallel with minimal cross shard communication
- New shards are created with increasing number of available nodes (i.e. increases capacity with increased resources)



Rethinking the Trilemma: Parallel Execution

Parallel Execution Motivation: Conflict Free Replicated Data Type (CRTD) generalize the concept of simultaneous distributed data transactions.

SagaChain uses a more limited version of the CRDT concept. It can be summarized as the following:

- Introducing the organization principle: object state database and the accounts as the organizing principles.
- An object in the object state database may only be modified on one shard at any given time. To enforce this, each shard has the concept of object ownership based around the account concept.



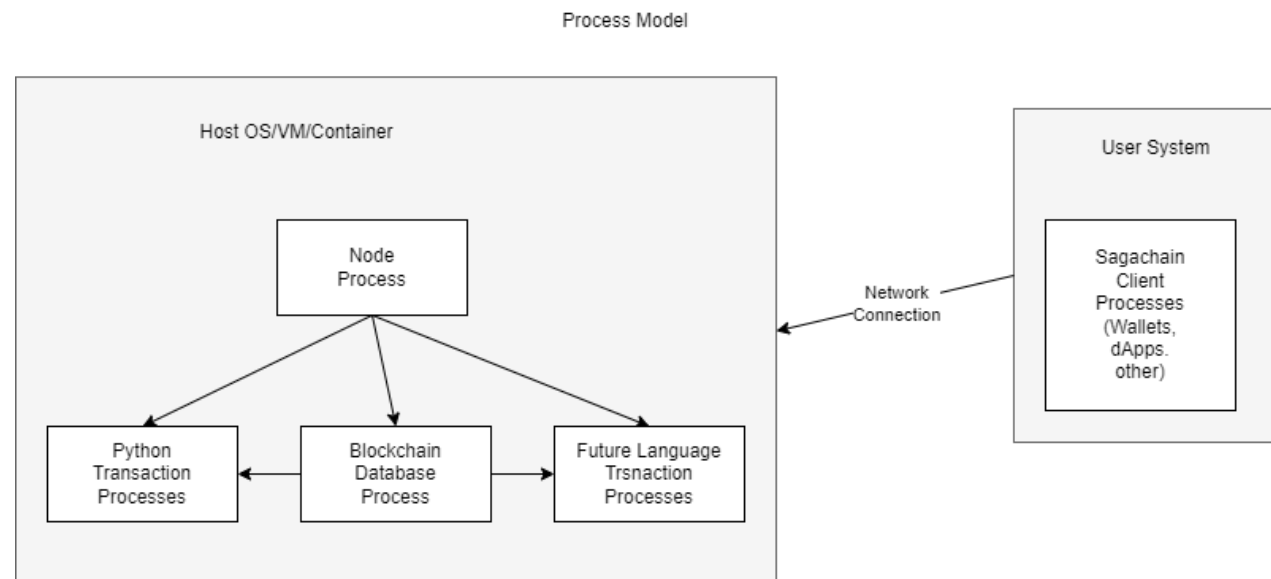
Open Motivation Requirement

- All code is open source for validation and mining
- All protocols are independent of specific software implementations



Introducing the Object State Database And Class Manager Infrastructure

Basic Process Concept

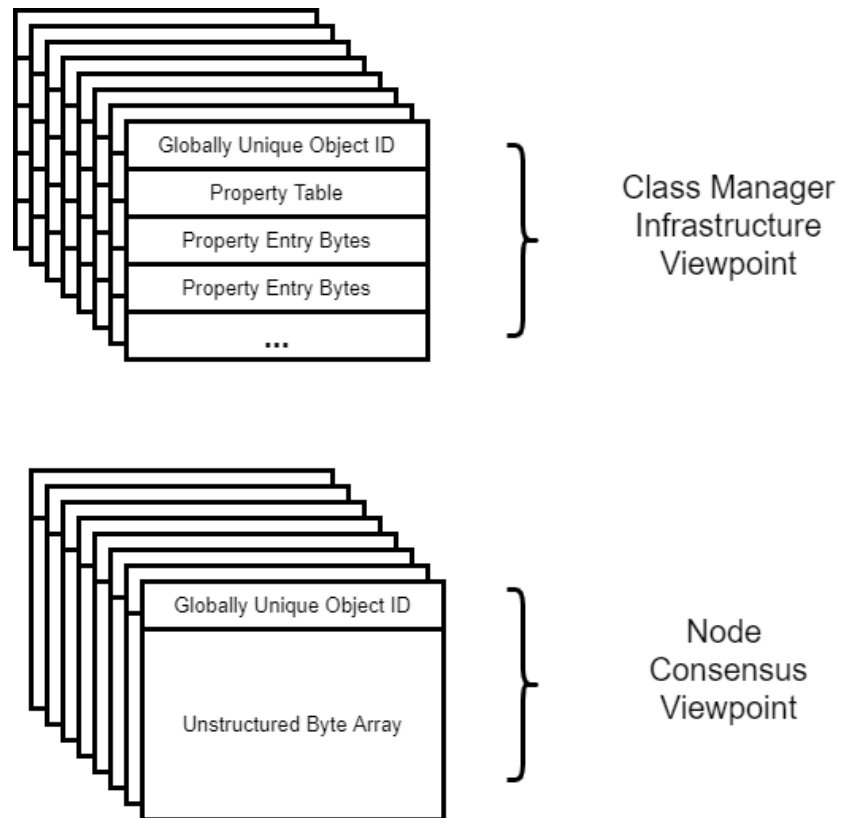




SagaPython Terms

- CMI – Class Manager Infrastructure
- CMI Object – an object's representation from the CMI viewpoint
- SagaPython - modified Python environment for SagaChain CMI
- SagaPython object – A CMI object's representation from the SagaPython viewpoint
- Python object – any Python object that is referenceable internally within a CMI SagaPython class
- SagaClass – CMI class created with modified Python syntax
- ClsObjVar – SagaPython class that represents a reference to a CMI object
- SagaClass Method – Method called by CMI method dispatch, written in SagaPython
- SagaClass Field – Data field visible to the CMI supporting get/set operations

Decentralized Object State Database





Object State Database Replication Consensus Viewpoint

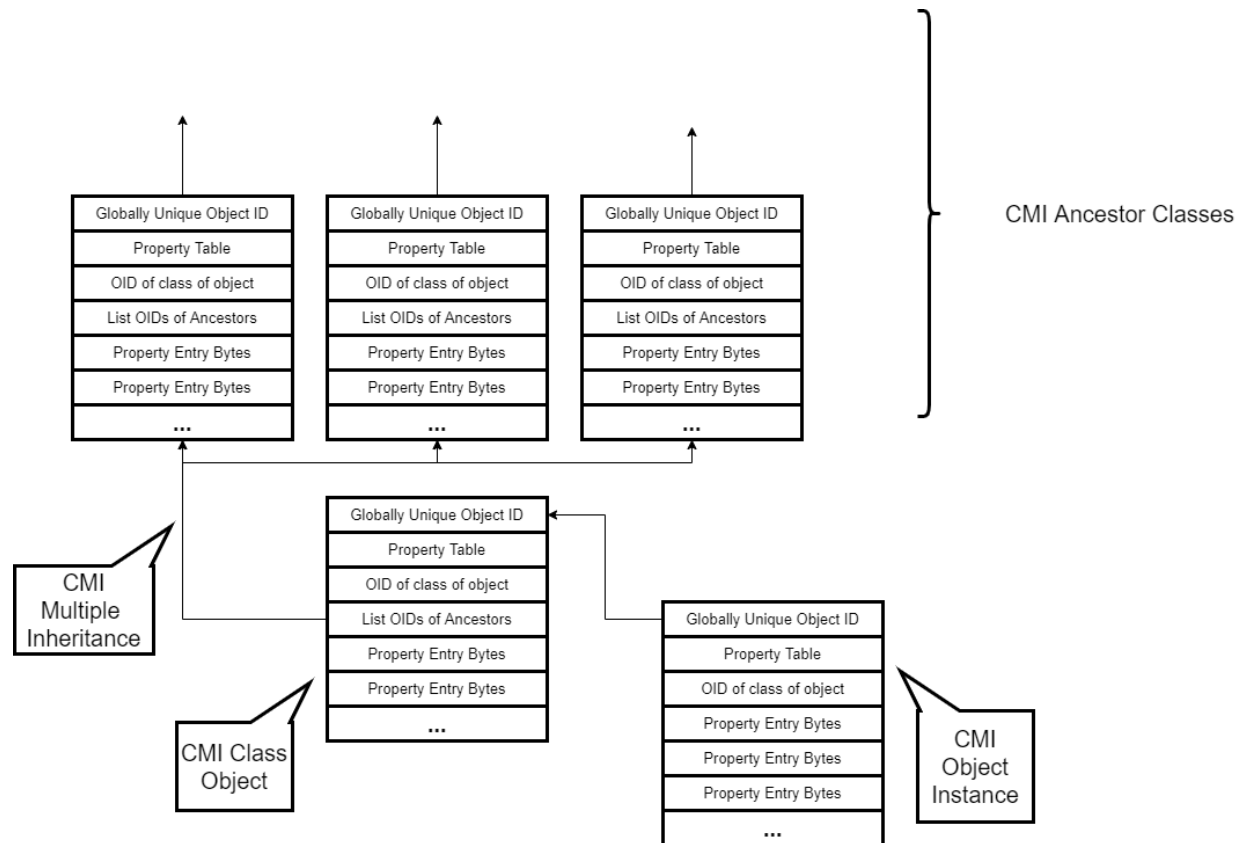
- Primary responsibility of the SagaChain is to ensure that the object state database has the most current copy of any given object when it is accessed on a shard.
- Object state database updates are strictly the result of transaction executions
- Consensus viewpoint consists of the transaction script and changes to any related objects in the object state database
- All objects are referenced by globally unique object identifiers: OID. These are 256 bit keys.



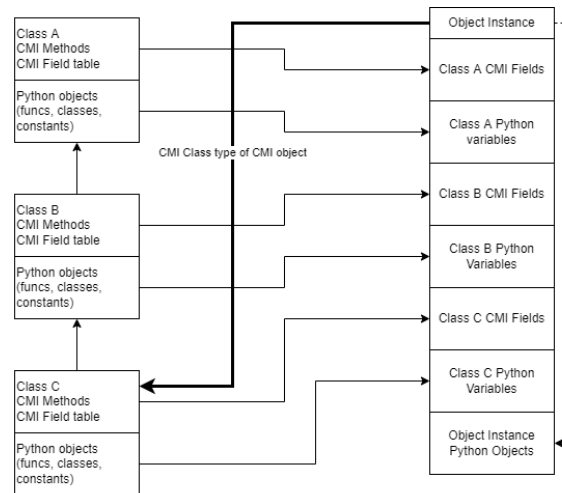
Object State Database Class Manager Infrastructure Viewpoint

- Each object consists of a table of properties
- The first property is the class type of the object
- All objects have a class type.
- Classes are also objects in the object state database.
- All method code for a class is stored in the object state database with the class
- All object instances in the object state database for a class reference the same class object in the object state database.
- Object state transitions is strictly a result of method or field access on an object instance of a given class type
- The Class Manager Infrastructure implement inheritance as references to other objects in the object state database as ancestor classes.

Class Manager Infrastructure Objects and Inheritance



SagaPython Classes, Objects and Inheritance



At CMI class creation, the Python functions, classes, and constants are created and stored in the class object properties. Direct access to Python objects is only through SagaPython runtime. That is, the CMI does not recognize the properties for SagaPython.

All CMI Class Definitions are in Transactions Scripts

SagaPython Object Instances can be of SagaPython class types that are of any CMI class.

CMI Class Methods and Field Definitions are Immutable

Logical Types of Access:
 CMI Methods
 CMI Fields
 CMI Class Python Funcs
 CMI Class Python Vars
 CMI Class Python Classes
 CMI Object Python Funcs
 CMI Object Python Vars
 CMI Object Python Classes
 CMI Object Python Objects

The background is a solid blue color with a complex, abstract pattern of overlapping shapes. On the left side, there are several concentric circles of varying shades of blue. Overlaid on these and the rest of the background are various geometric shapes, including rectangles and trapezoids, some of which are oriented diagonally. The overall effect is a layered, modern aesthetic.

Account Model and Global Invariant



Organizing Principle: Invariants

- Account objects are the organizing principle
- All objects are “owned” by account objects
- Invariant: An account object and any objects it owns may only be modified on one shard at a time.
- Invariant: When an account object and any owned objects are modified, the precondition assertion – the state of the objects are globally upto date at that time.
- Creates the concept: Account ownership by individual shards.
 - Corollary: the shards negotiate account ownership, not object state. Eliminates synchronized global object state update.



Organizing Principle: Account Ownership Negotiation

- Shards are assigned account ownership using a cross-shard negotiation and consensus
- Accounts may be in operational states:
 - Currently owned by a shard
 - Unowned by any shard
 - Ownership transitioning
- Shards negotiate account ownership, not object state
- Object state is gossiped globally independent of account ownership
 - Shards may request object state of an account from the previous shard owner on direct shard-to-shard account ownership transfers
- Account ownership is primary means for load balancing across shards

The background is a solid blue color with a complex, abstract pattern of overlapping circles and geometric shapes in various shades of blue, creating a sense of depth and movement.

Python Backgrounder



Python Programming Language

What is Python?

“Python is a programming language that lets you work more quickly and integrate your systems more effectively.

You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs. ” – from the python.org website

Python is an open source, dynamically compiled, bytecode interpreted, procedural, object-oriented and functional programming language. It has more than a 20 year history, and currently is on release version 3.10.5.

Some companies using Python: Google, Youtube, [Industrial Light & Magic](#), [EVE Online](#)



Python Programming Language Code Fragment Examples

Comments are preceded by a “#”

#defining a function

```
def afunc():
```

```
    ...
```

#defining a class

```
class aclass:
```

```
    def method1():
```

```
        pass
```

#calling a function

```
Reval = afunc()
```

#control flow

```
if X > 10:
```

```
    # do something here
```

```
elseif X <= 10:
```

```
    # do something else
```

```
else:
```

```
    # do this
```

#assigning a variable

```
avar = 'this is a string'
```

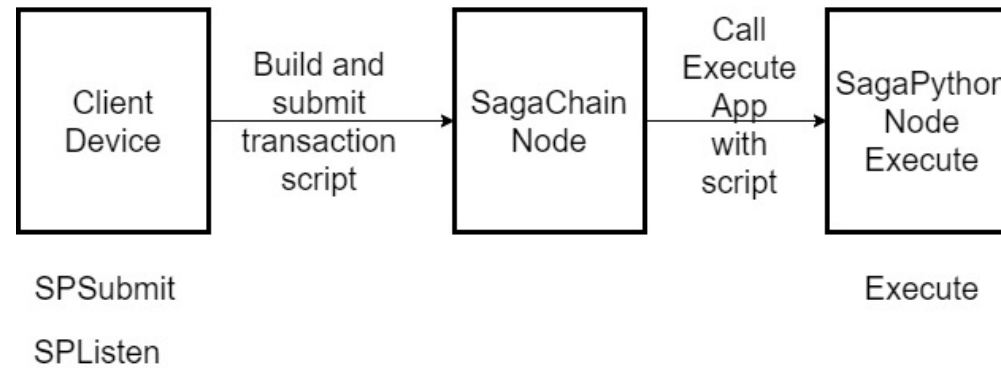
Maxim: “Everything is an object in Python”



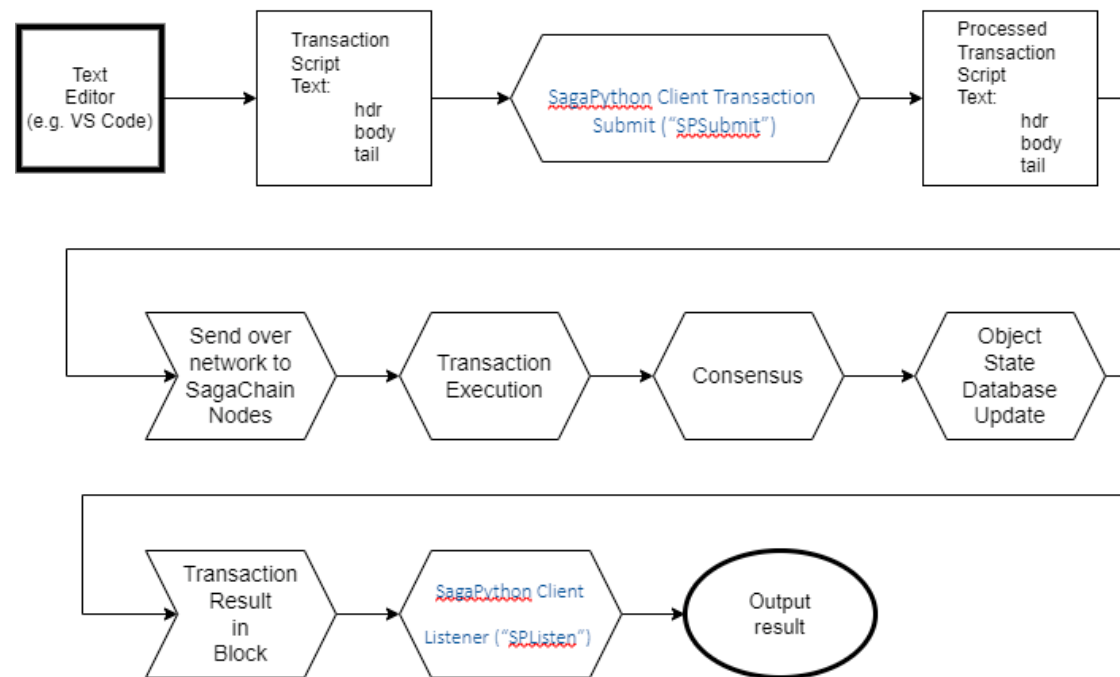
Python Programming Language Used for SagaPython

- SagaPython uses syntax consistent with Python. The execution environment is restricted. Not all Python code will execute in SagaPython.
- SagaPython uses the dynamic compiler and interpreter of Python. Nodes load class code from the object state database dynamically, compiling locally to bytecode and interpret the bytecode with the Python interpreter
- The SagaPython interpreter adds bytecode counting for gas consumption to the execution environment
- SagaPython integrates the Class Manager Infrastructure (CMI) class model and object state database, with the Python procedural, functional and object-oriented environment.
- All user SagaPython code executed on SagaChain is stored in the object state database

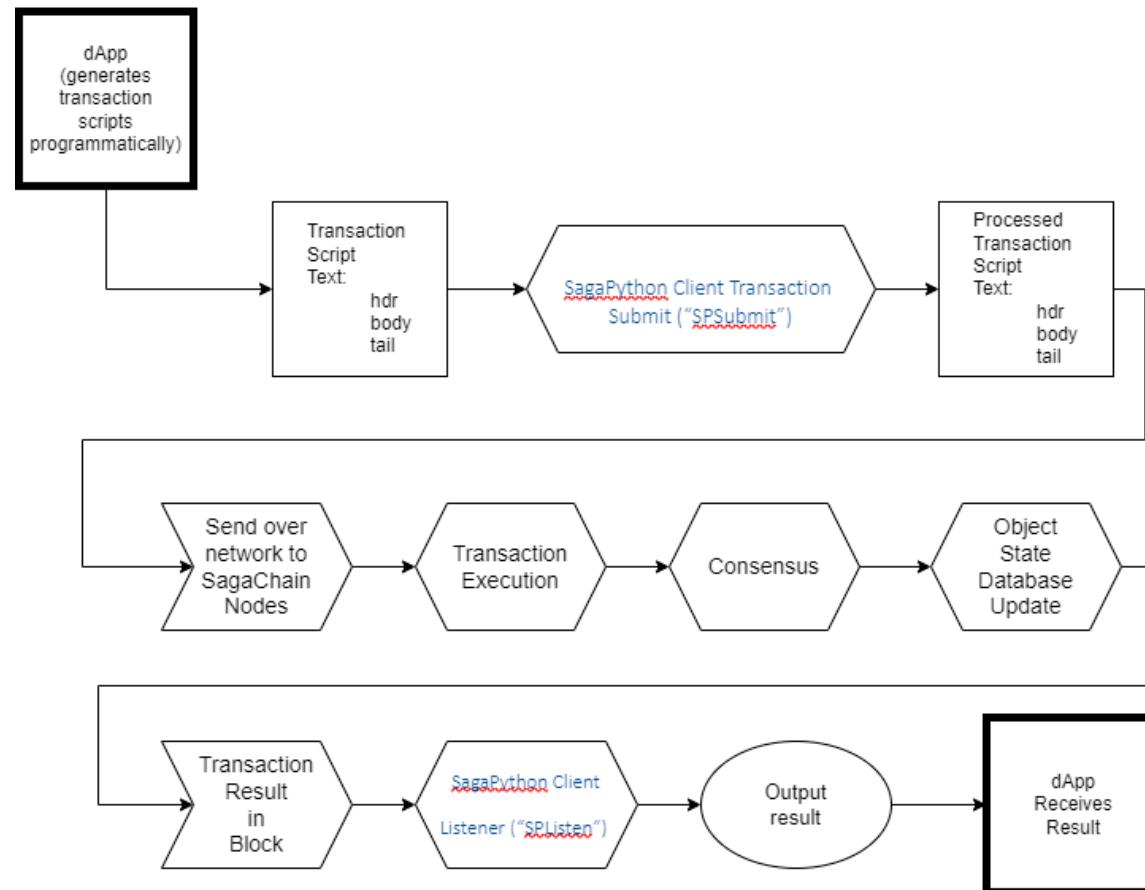
Generalized Transaction Script Flow

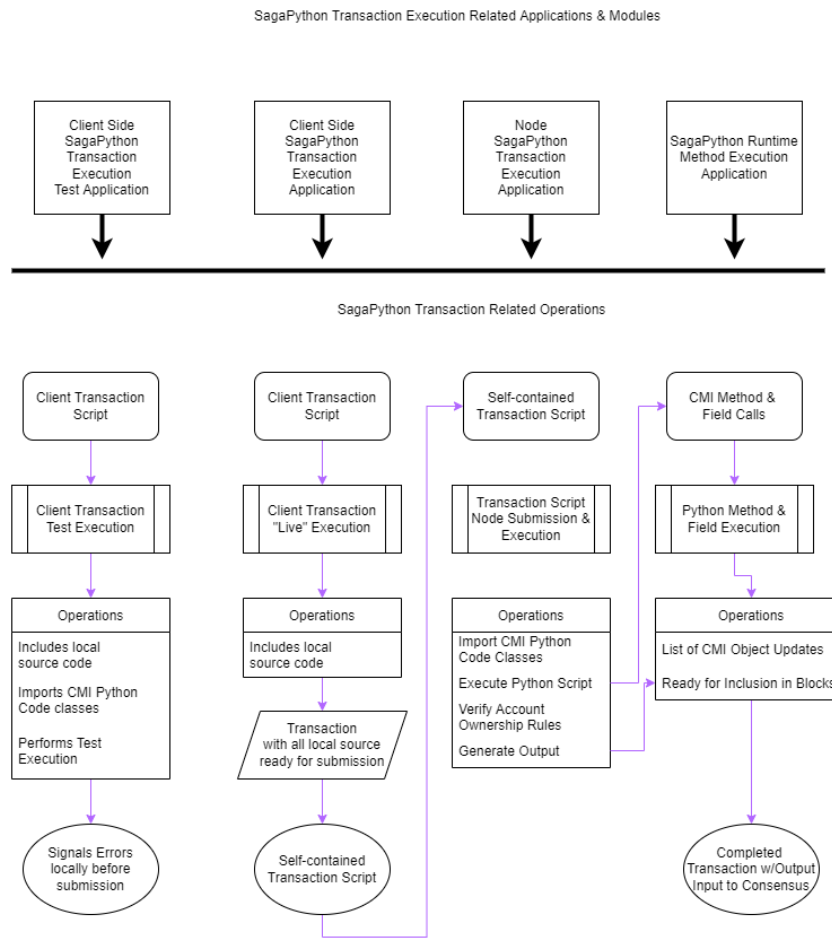


Transaction Script Flow from Manual Text Editor



Transaction Script Flow from Client dApp





Transaction Script Execution Flow

SagaPython Applications take SagaPython source code as input. Internally the code is compiled to Python bytecode dynamically, executed, and may be cached by a node for internal performance. Logically, all SagaPython code is executed from source code. SagaChain logical object state database and Blockchain database store SagaPython source code. Internally a node may store SagaPython as Python compiled code transparently to SagaChain operations, network and external interfaces.

SagaPython source code is distinguished from Python source code as executable by the SagaPython applications diagrammed above.



SagaPython Client Command Line Tools



Client Application: SPSubmit

- SPSubmit <transscriptname>, <privkeyfile>, keys = [{“toacct”:”12345”: targprivkeyfile}],
-D targ1 = “...”, -D targ2 = “...”
- SPSubmit takes the following parameters:
 - TransactionScript
 - Client Account Private Key
 - Array of Tuples
 - Keys = [{<varname>:<OID>:<private key>}, ...]
 - Variable name definitions: -D <Varname> = <value>
 - Optional output transaction filename, if present only a file is generated, but not submitted
 - -o <filename>
- Returns:
 - The hash of the transaction



Client Application: SPSTListen

- SPSTListen “FACE123”, 6456, “<32 byte hash>”, 6
- SPSTListen takes the following parameters:
 - Account OID
 - Sequence number
 - Hash value returned from SPSTSubmit
 - Confirmation Count



SagaPython Transaction Script Overview



SagaPython Decorator Syntax

Python class decorator syntax is defined in “PEP 3129 Class Decorators” :
<https://peps.python.org/pep-3129/>

SagaPython class decorator is `SagaClass()`:

`@SagaClass(base class list, /, metaclass = ..., account = ..., oid = ..., name = ..., imports = [class code import list])`

- **Base class list** supports multiple inheritance with a method resolution order (MRO) that follows the same C3 MRO as Python, for CMI class inheritance. Algorithm is described in, “A Monotonic Superclass Linearization for Dylan”, published June 28, 1996



SagaPython Decorator Syntax

@SagaClass(base class list, /, metaclass = ..., account = ..., oid = ..., name = ..., imports = [class code import list])

- **metaclass:** enables overriding default CMI metaclass
- **account:** specify account context to create new class in, default is current account
- **oid:** specify explicit object ID for new class, default class creation generates the OID
- **name:** registers a string name for the class with the global object name dictionary
 - high overhead, default is unnamed, referenced only by OID
- **Imports:** defines a list of ClassModule object instances that contain Python module source code for internal import to the new CMI class. This is distinguished from CMI class inheritance.



SagaPython

Key Syntax Components



SagaPython Transaction Script

```
def __hdr():
```

```
...
```

```
def __body():
```

```
...
```

```
def __tail():
```

```
...
```

- Each function is compiled logically as a separate program

- `__hdr()` contains transaction header information (e.g. gas units)

- `__body()` contains the transaction code

- `__tail()` contains a transaction hash and client account signature, generated by SPSubmit



Transaction Script Header

```
def __hdr():  
    return {  
        'acct': <account object ID>,  
        'seq': <transaction sequence number>,  
        'maxGU': <gas units>,  
        'feePerGU': <gas unit price>,  
        'extraPerGU': <extra gas unit price>,  
        'extraData': <unstructured bytes – opt.>  
    }
```

- `__hdr()` - returns a dictionary with the listed keys and values.
- `acct` - CMI object state database ID for the account object verified by the client's signature
- `seq` – incremented transaction sequence number for the account
- Gas parameters – based on transaction bytes and execution
- `extraData` – unstructured bytes referencible in the `__body()` function



Transaction Script Body

```
def __body():
    #general python code
    # Python variables | python functions
    #example:
    x = 1
    name = "python text"

    #Sending a SagaPython messages:
    # creates a proxy object for the object instance
    # referenced by <objectID>
    objvar = ClsObjVar(<objectID>)

    # example method call on the object, with a method named
    "method1"
    retval = objvar().method1(x, name)

    # mix of Python code for control flow
    # and SagaPython messaging

    if retval == <some value>:
        # send another message to a different object
        objvar2 = ClsObjVar(<objectID>)
        sndretval = objvar2().<another method>(<arg list>)
    else:
        # do something else here

    # any additional amount of Python control flow,
    # and SagaPython messaging

    # logging may be explicit with the Log() function
    Log (<transaction output values>)
```

`__body()` is compiled and executed by the SagaChain nodes, not the client SPSubmit application

The `__body()` function contains all control flow and messaging logic. Messages are sent to CMI objects by creating an instance of `ClsObjVar()`, with the object ID of the target object, and sent using the function and “dot operator” notation: `objvar().method1(args)`.

As is shown, there can be any amount of control flow code as well as multiple messages to different methods and different objects. A transaction must finish without an execution error, fully as a unit. No state changes occur for partial execution.



Transaction Script Tail

```
def __tail():  
    return {  
        'hash': <hash of transaction>,  
        'sig': <signature of hash>  
    }
```

- `__tail()` - returns the hash and signature for the transaction. The two fields, 'hash' and 'sig' are required. The values are ignored and overwritten by the SPSubmit application
- hash is the SHA256 value of the bytes of the text of including the functions `__hdr()`, `__body()`, and additional text added by the SPSubmit command
- sig is the signature of the hash with the client's account's private key to verify the source of the transaction



Creating A New Account



Class Account

- Contains:
 - Transaction Sequence Number
 - SagaCoin balance
 - Public Keys
 - Child Objects

Transaction Script Account Creation

```
# transaction script to create a new account
# Creating a new account by sending a transaction
# to an account manager object that acts as a
# factory object.
# The account containing the account manager must
# publish its private key as a wellknown key for
# this use. Not advisable for general accounts.
def __hdr():
    owningaccount = 'ABCDE'
    # normally a single global wellknown account for
    # creating accounts
    transactionseqnum = 0
    # special seq number when transaction ordering is
    # unimportant
    hdr = {
        'acct': owningaccount,
        'seq': transactionseqnum,
        'maxGU': 10,
        'feePerGU': 1,
        'extraPerGU': 2
    }
    return hdr
# must be a dictionary with the transaction header values
# does not return to the client, return is used here as a
# convenience for transaction execution only
```

- Standard header portion of all transactions.
- Global account factory with a well known ObjectID. Creates new accounts of type Class Account
- New account factory instances can be created to create different accounts from subclasses of type Class Account.
- ClassAccountFactory itself may be subclassed and a new instance created for creating accounts



Transaction Script Account Creation

```
def __body():  
    accmgr = ClsObjVar('bbbb')  
  
    newacct, err = accmgr().MakeAcct()  
    log(newacct.oid)
```

```
# the body of the transaction sends a  
# message to the account manager to create a  
# new account  
# object ID of the account manager object  
# which  
# of owningaccount  
# returns a ClsObjVar instance for the new  
# records the oid in the transaction log in the  
# transaction do not "return" values  
# must listen to blockchain for confirmation
```



Transaction Script Account Creation

```
def __tail():  
    return {  
        'hash': 12345,  
        'sig': (rvalue, svalue)  
    }
```

tail contains hash of __hdr and __body functions.

tuple of r and s value for signature

Transaction Script SagaCoin Transfer



Transaction Script SagaCoin Transfer

```
def __hdr():  
    hdr = {  
        'acct': 'aaa',  
        'seq': 1000,    # next sequence number  
        'maxGU': 10,  
        'feePerGU': 1,  
        'extraPerGU': 2  
    }  
    return hdr
```

```
# Sending sagacoin from current account to  
# another  
# transaction is from curracct to 'bbb'  
# curracct is the ClsObjVar instance for 'acct'  
# the signature on the transaction script  
# verifies the caller.  
# No verification on the receiver.  
# Anyone can send SagaCoin to the receiver
```



Transaction Script SagaCoin Transfer

```
def __body():  
    toacct = ClsObjVar('bbb')  
    coins, err = curracct().take(NNN)                                # subtracts amount from the current account  
    if coins != NNN:  
        ErrorExit(err)  
  
    err = toacct.put(NNN)  
    if err != None:                                                  # adds NNN coins to account referenced by  
        ErrorExit(err)                                              # toacct
```



Transaction Script Creating a Class Example

ClassAsset

Transaction Script Creating Class Asset

```
def __body():
    @SagaClass(ClsSagaClass,metaclass=ClassClass,account='Global')
    class ClsAsset:
        SagaFieldTable=[]

        def __init__(self, name: str, initcount: int):
            self.assetcount = initcount
            self.name = name
            self.callcount = 0

        @SagaMethod()
        def Increment(self, inc: int):
            self.assetcount += inc

        @SagaMethod()
        def decrement(self, dec: int):
            if self.assetcount - dec > 0 :
                self.assetcount -= dec
            else:
                raise RuntimeError("negative
                asset counts are illegal")
```

internal method - example simply counts number of calls

```
def callcount(self):
    self.callcount +=1
```

The name ClsAsset is only available to the transaction script
The objectID must be retrieved if the intent is to use the class
oid = ClsAsset().oid

log it for user to recover it, could also store it in another object
Log(oid)

create a reference to the class object
classvar = ClsObjVar(oid)
An instance of the new class can be instantiated
initialized with 100 count of asset
objvar = classvar().New(name, 100)

make the objvar persistent by inserting it in the account list
also sets owner field of objvar
curracct().Insert(objvar)

Log(objvar().oid)



Transaction Script 3rd Party Transfer Between 2 Accounts



Transaction Script 3rd Party Transfer Between 2 Accounts

- Three parties are involved in a 3rd party transfer
 - Sending party
 - Receiving party
 - Orchestrating party
- The orchestrating party has custody of a key for each party
 - Writes a transaction script for the transfer for each party
 - Generates signatures using custodial keys from the parties
 - Signs transaction with it's own key
- Unless orchestrating party modifies its own objects, does not require the orchestrating party state update
 - Can do multiple transfers on multiple shards simultaneously



Transaction Script 3rd Party Transfer Between 2 Accounts

```
# Header signatures are generated by the client transaction builder.  
# The signatures are included as variables.  
# They are a signature of the hash of the hdr.  
# Because the header includes a sequence number, a replay  
# attack with the signatures isn't possible.  
  
# Assume that there are two accounts, and that the transaction  
# is to transfer funds between these two accounts:  
# Signature variable names are: FromAccountSig and ToAccountSig  
# Further, that there are two OID definitions passed in as well as variable names:  
# FromAccount, ToAccount, and TransferAmount as the amount to transfer
```



Transaction Script 3rd Party Transfer Between 2 Accounts

```
def __hdr():  
    hdr = {  
        'acct': 'aaa',  
        'seq': 1000,  
        'maxGU': 10,  
        'feePerGU': 1,  
        'extraPerGU': 2  
    }  
  
    return hdr
```

The account and sequence number are for
the 3rd party, which signs the transaction



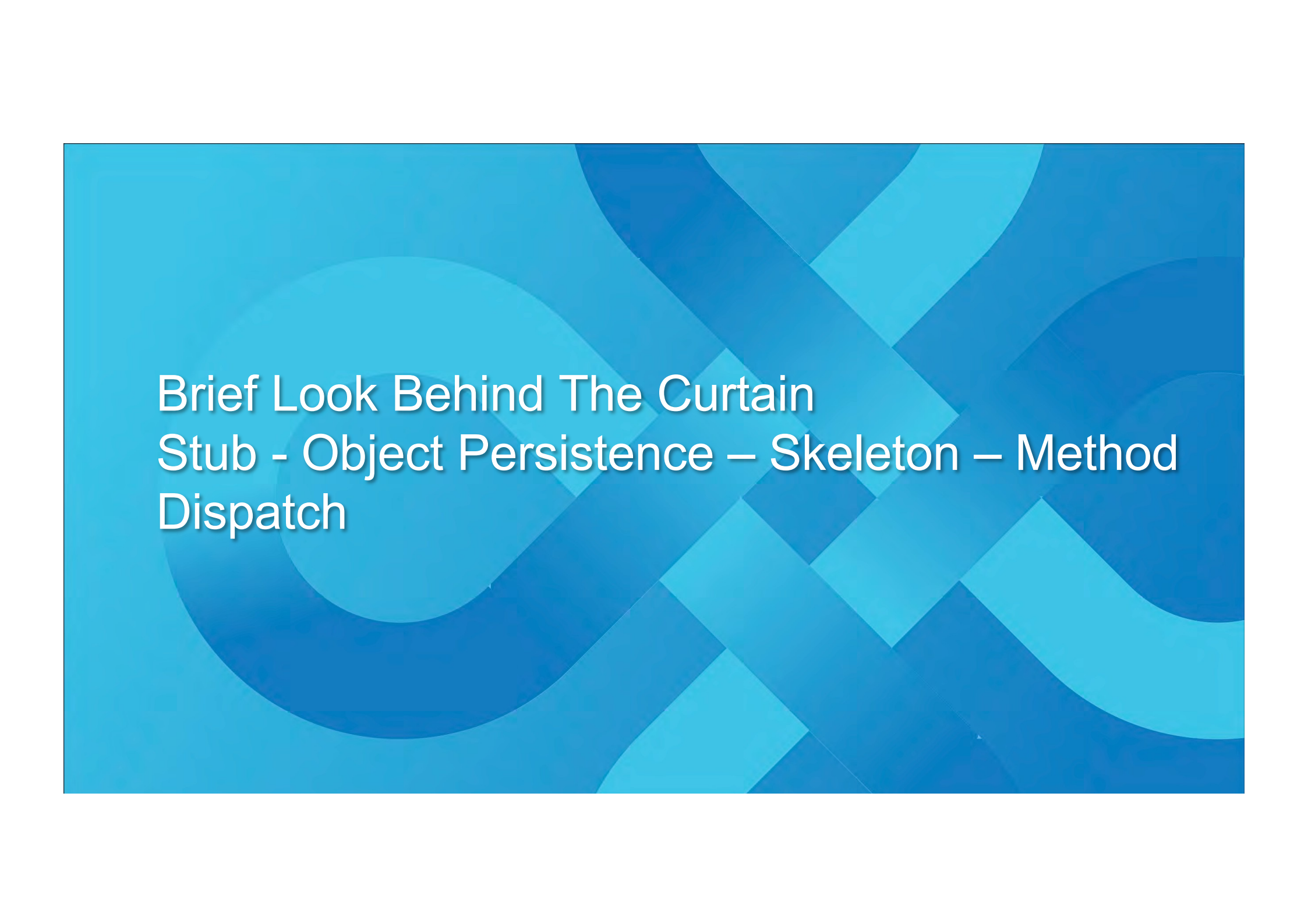
Transaction Script 3rd Party Transfer Between 2 Accounts

```
def __body():  
    from = ClsObjVar(FromAccount)  
    to = ClsObjVar(ToAccount)  
    coins = from().take(TransferAmount, sig =  
        FromAccountSig)  
    if coins != TransferAmount:  
        Log(error)  
        ErrorExit()  
  
    err = to().put(TransferAmount, sig =  
        ToAccountSig)  
    if err != None:  
        Log(error)  
        ErrorExit()
```

The account and sequence number are for
the 3rd party, which signs the transaction

#FromAccount, ToAccount are passed in OIDs,
#from SPSubmit commandline, or could be
#explicitly written.

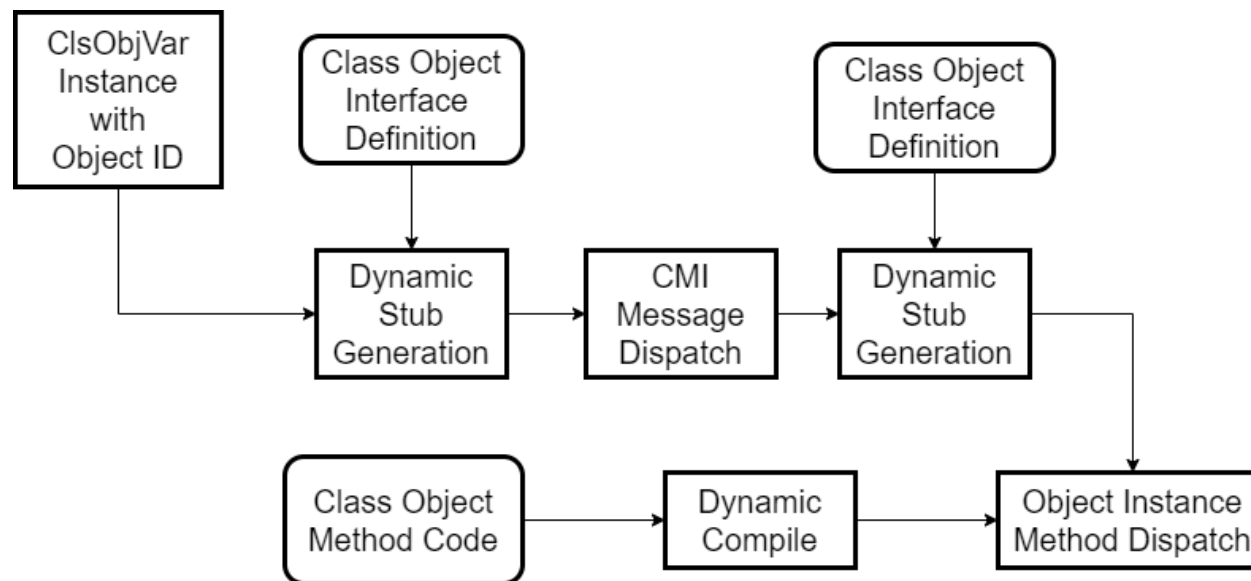
#FromAccountSig, ToAccountSig use the hash
#of the __hdr() to authenticate both accounts



Brief Look Behind The Curtain

Stub - Object Persistence – Skeleton – Method Dispatch

Dynamic Stub and Skeleton Dispatch Flow





Useful Python Features

- Object Instance Callables
 - Defines the `__call__()` method in a class
 - Function syntax used with the object instance, (i.e. `<objectvariable>()`) calls the `__call__()` method
 - Python docs link: <https://docs.python.org/3/reference/datamodel.html?highlight=callable>
- Object attribute access (both methods and fields)
 - Defines the `__getattr__()` method in a class
 - Called unconditionally to implement attribute accesses for instances of the class.
 - Give complete control of how the “dot” operator syntax is executed
- Class Decorators
 - Defines functions that begin with the “@” sign, and followed by a class definition
 - Enables creating the CMI class in the object state database, leveraging Python introspection features
- Import implementation
 - `sys.meta_path` enables module imports from the objects in the object state database
 - `sys.path` – disable local file importing by clearing `sys.path`



Python Runtime Modifications

- Byte counting added to interpreter loop for gas units
 - CMI message dispatch internals does not consume extra gas units, encouraging inheritance
- Importing of Python modules by URL, file and pathname or any other approach, other than from objects in the object state database is disabled
- Python modules that interact with the underlying OS are removed from the Python runtime environment



ClsObjVar to Stub Transform: Python Feature

- Syntax:
 - `result = objvar().method(arguments)`
- ClsObjVar object instances are callables
- `objvar()` syntax internally returns a stub object
- dot syntax is applied to the stub object, not the ClsObjVar object instance
- stub object uses `__getattr__()` Python method to capture the method and field accesses
- `__getattr__()` method implements the interface to the CMI, and returns result

Result from the calling code viewpoint is a normal Python interaction, aside from the requirement for the “()” extra syntax



Skeleton to Method Dispatch Transform: Python Feature

- CMI Class defined method is invoked with the object instance data in the “self” object
- The passed in self object, instead of being the usual Python object instance is replaced with the skeleton object as a proxy to the object instance
- `__getattr__()` is used such that the developer’s code uses the self object in the normal manner transparently
- `__getattr__()` interface with the CMI to load and internally dynamically compile the class methods and performs method dispatch as though it is a `ClsObjVar` object, providing common syntax using the self object with `ClsObjVar` objects.
 - The skeleton class implements `__call__()` callable to match syntax



Prototype For SagaPython Transaction Execute Application

load sagapython body objects here

with open(sys.argv[3]) as sf:
sagascript = sf.read()

sagaobjs = compile(sagascript, sys.argv[3], 'exec')

if sagaobjs == None:
sys.stderr("Failed to compile Sagapython bodywrapper
environment")
return -1

bodyglobals = sagaglobals.copy()

try:
exec(sagaobjs, bodyglobals)
except Exception:
err = "SagaPython bodywrapper functions load
failed"
sys.stderr(err)
return -1

now have the body wrapper function in
bodyglobals

need to load the body function -

try:
exec(bodyobj, bodyglobals)
except Exception:
sys.stderr("Failed to load __body() function")
return -1

call the body wrapper function
bodyinfo = eval("bodywrapper()", bodyglobals)

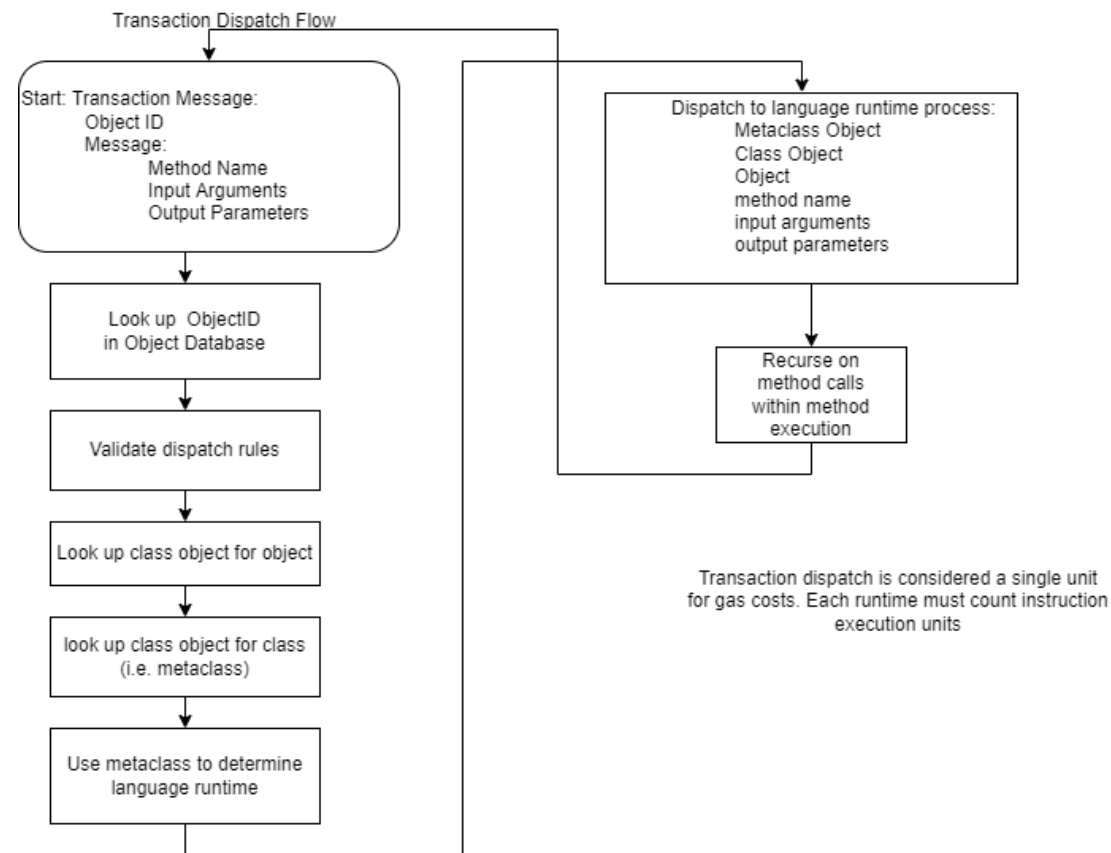
if bodyinfo == None:
sys.stderr("Failed to run body function")
return -1



Source Code to Examples

- URL: <https://www.xbom.io/> to register for access to the examples repository and copy of the latest SagaPython syntax doc
 - Bodytest.py
 - Newaccountexample.py
 - Classassetexample.py
 - Sendsagacoin.py
 - From-to.py
 - Sagapythontransaction.py

Meta Transaction Dispatch Flow







ReImagined.

SPSListen

The me traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated.

The fire burned brightly, and the so radiance of the incandescent lights in the lilies of silver caught the bubbles that flashed and passed in our glasses.

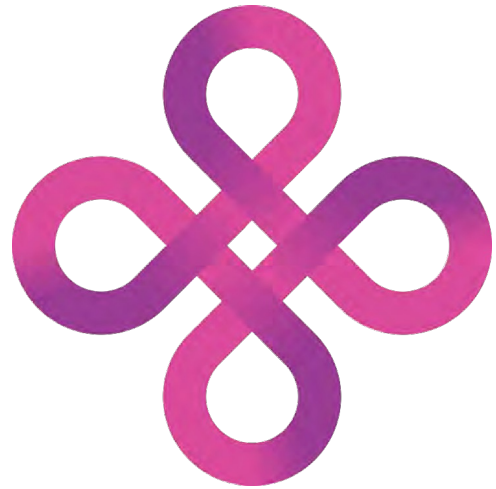
Our chairs, being his patents, embraced and caressed us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere .

Revolutionizing industry.

The me traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated. The fire burned brightly, and the so radiance of the incandescent lights in the lilies of silver caught the bubbles that flashed and passed in our glasses. Our chairs, being his patents, embraced and caressed us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere when thought runs gracefully free of the trammels of precision. And he put it to us in this way – marking the points with a lean forefinger – as we sat and lazily admired his earnestness over this new paradox (as we thought it) and his fecundity.



Purpose.



PraSaga™