SagaPython User Guide

Author	Description	Version	Date
David Beberman	Incomplete 1st draft	0.01	20221203
“”	Added additional examples	0.02	20221204
“”	Added reference section. Finished examples. Argument types tbd	V 0.9	20221206
“”	Added to reference section. A V1.0. Section Argument types remains tbd	V1.0	20221207

Table of Contents

Introduction

The following sections use explanation of examples to demonstrate SagaChain transaction scripts written with SagaPython. The two main tools for SagaPython: sptransactionexecutor.py; and spclient.py are demonstrated with the commandlines for each for each example. The examples may be found in the testtransactions/ directory of the SagaPython source pool.

For each example, relevant lines are numbered. The following texts provide a line-by-line explanation for each example. The examples covered include:

- Creating an account
- Sending SagaCoin between accounts
- Creating a new class and object
- Sending a message to the new object,
    - which executes a method defined by the new class

Overview

Object State Database

SagaChain and SagaPython are oriented around creating, maintaining, and accessing objects in the SagaChain Object State Database. It will be referred to as the database throughout this text.

The database is flat keyword and data. The keyword values are globally unique 256 bit values. These are known as a ledger object identifiers ("LOID"). The data is a bytearray of varying length. The database supports standard creation, reading, writing and deletion. The current implementation uses LevelDB. From the viewpoint of SagaPython the database implementation is transparent and may change.

At its simplest form, an object retrieved from the database is a contiguous array of bytes. The SagaChain class and object model, called the Class Manager Infrastructure, ("CMI"), is implemented as layers that interact with the database. The CMI presents the class and object model to the SagaPython environment.

CMI Environment

The CMI environment uses message passing. All interactions with objects in the database are through messages to interfaces defined as methods and fields. The methods and fields are defined and implemented by classes. The classes are objects stored as LOID keyed byte arrays from the point of view of the database. Object instance state is stored in objects that reference their class using the LOID of the class. The CMI supports multiple inheritance using a class inheritance linearization algorithm called C3.

SagaPython adds some additional features specific to a Python environment while adhering to the method and field interface model of the CMI.

General Usage Overview

All interactions with the database are via transactions sent to the SagaChain. Transactions may include the following:

- Account creation and deletion
- Class creation*
- Object creation and deletion
- Object messaging
- General purpose control flow

*class deletion is not supported

(the alpha version of SagaPython simulates submission to the SagaChain by the sptransactionexecution application)

From a user's point of view, interaction with SagaChain can be generalized to the following operations:

- Account Key Generation
- Account Creation
- Transaction Writing
- Transaction Signing
- Transaction Submission
- Receiving Results

Account Key Generation

Generate a private signing key and public verification key with the edsig tool. This generates a private signing key stored as 32 bytes, and a public verification key store as ascii hex digits. Store the private key. The public key is used in creating the account object.

The commandline for edsig:

edsig generate <prefix for generated key files>

The output files are <prefix>signingkey and <prefix>.verifyingkey

If no parameters are provided to edsig it prints out commandline help.

There is a global well known signing key: systemaccount.signingkey. This is used as a bootstrap to create new accounts.

Account Creation

Creating a new account consists of sending a transaction to an object using a design pattern known as a factory object. The foundation classes for the CMI include a class called SPClassAccountFactory and a foundation object instance called SPAccountFactory. New accounts may be created by sending the message new to the SPAccountFactory.

- A new account creation transaction must be signed and sent to an existing account in the database. The SystemAccount serves this purpose.
- The spclient application signs the transaction with a signing key. The systemaccount.signingkey serves this purpose.
- The accts field in the header section includes the SystemAccount LOID as one of the listed accounts.

- The body section sends a message to the SPAccountFactory which is an object owned by the SystemAccount.
- The message contains the public verifying key that the user generated for the account.
- The SystemAccount public key stored in the SystemAccount object verifies the signature for the transaction.
- The SPAccountFactory then creates a new account object, storing the public verifying key in the new account object for future verification and returns the LOID.
- The user stores the LOID for future transactions. To send a transaction to an object owned by the account, the account is listed in the transaction header, and is signed with the signingkey that the user stored previously.

Transaction Writing, Signing, Submission, Results

A transaction consists of messaging operations and general purpose control flow code. A transaction consists of four sections: header; cmiclasses; body; and tail. The syntax of the components of a transaction are described in the examples below with line-by-line explanations.

The user is responsible for the three sections: header, cmiclasses, and body. The signing application, spclient is responsible for the tail component.

The general flow:

- user writes a new transaction with either a text editor or a (future) 3rd party tool
- user signs the transaction with the spclient application
- user submits the transaction to SagaChain (simulated with sptransactionexecutor application)
- user collects results of the transaction

The two main tools are spclient and sptransactionexecutor. The reference provides the commandline switches for these. The general formats are:

python3 spclient.py <input transaction> -s "<quoted list of comma separate key file names>", -o <output signed transaction file>

Note: The key files must be in the order that they are listed in the transaction header.

python3 sptransactionexecutor.py <input signed transaction> -s <database name> -o <results text file>

The database may be reused in future transactions as it contains the accumulated state of the objects of the transactions. As the foundation classes and objects are unique and any created objects are derived directly or indirectly from the SystemAccount object, the database may be shared among developers.

If no database is provided it will default to 3 databases: input database, read accessed objects database and read/written objects database. This mode is intended for use with the SagaChain nodes. It does not update the input database which may cause some confusion during development.

All results are output explicitly using the **Log()** function. This writes directly to the transaction text file using standard printable strings. This is not meant as a debugging tool. Suggest using debugging print statements to stdout and stderr in the transactions and any development/test classes.

Example Transactions

(Line numbers are for documentation, and are not part of the transaction scripts)

Prerequisites

Signing Key File: systemaccount.signing.key

The signing key is stored in bytes format.

Verifying Key File: systemacccount.verifying.key

verf0-mzaq6yjm4cir6jdocltctskqvr2xcuf4ixxeyzqehyenxm5rt7tq

The verifying key is stored in ascii hex format for useability.

SystemAccount LOID: sysaccount.py

SystemAccount = bytes.hex(bytes.fromhex('AB CD EF EB 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'))

Defines the LOID string in ascii hex for the SystemAccount object.

New Account Creation Transaction Example

```
def __hdr():
1   sysacct = doimport('sysaccount.py')
2   CMIImport('sysacct', 'sysaccount', '*')


3   hdr = { 'accts': [SystemAccount],
4           'seq': 1000,      # whatever the next sequence number is
5                   'maxGU': 10,
6                   'feePerGU': 1.0,
7                   'extraPerGU': 2.0,
    }
```

```
 8  return hdr


def __body():
9    acctpk = 'verf0-jvrbxfdnejuwgewecpqnekcnbt7og37bmtfynd5ceucpr32irszq'


   # A reference to the accountfactory
10   factoryobj = ClsObjVar(CMIConst.SPAccountFactory)


   # create a new account with factory method
11   newacct = factoryobj.FactoryNew('200.00', acctpk)
12   Log("New Account Instance: ", newacct.oid)
13   return True
```

Explanation Line by Line

Line 1:

Doimport() is a function processed by the spclient application while signing the transaction script.

Doimport(<filename>) reads the contents from the filename directly in to the transaction script when the transaction script is passed as input to the spclient application. The output signed transaction will replace the Doimport(<filename>) text with the designated file in a zipped, base64 encoded format as a string assigned to the given variable, in this case, the variable is sysacct. The transaction hash is based on the output zipped, base64 encoded string, as opposed to the Doimport() syntax.

Line 2:

CMIImport('sysacct', 'sysaccount', '*') is a function processed by the sptransactionexecutor application and thus is processed by the SagaChain, unlike Line 1 above.

CMIImport('<variable name', '<module name>','<import symbols>') is roughly similar to the standard Python import syntax:

- The variable name passed in is the variable for the string that was assigned in Line 1.
- The string is decoded, unzipped, and converted back in to a Python file.
- The Python file is executed as a Python module.
- The Python module is given the name passed in as the <module name> string

- Symbols in the loaded Python module may be added to the global namespace of the currently executing component which is the __hdr() component in this case. The '*' syntax means load all the symbols. This is equivalent to:
    - from <modulename> import '*'

Lines 3 – 7 define the dictionary object that must be returned by the __hdr() function.

Line 3:

accts is the ordered list of accounts matching the signatures that will be attached as the tail component. The first account in the list is defined to be the owner of the submitted transaction. Additional accounts in the list with the attached signatures provide proof of key ownership.

Line 4:

seq is an incrementing number for each transaction submitted by the owning account. This value must be greater than all previous transaction sequence numbers for this account. The most recent sequence number is stored in the account object. The sequence number prevents replay attacks.

Line 5:

MaxGU is the maximum execution units allowed for completing the transaction execution. These are known as gas units in the industry. This is a simply integer number. There are no fractions of gas units.

Line 6:

feePerGU is the fee in SagaCoin that the user is willing to pay for the consumed gas units. SagaCoin is a decimal number with up to 8 places after the decimal and a maximum value of 20 places. This value is expected to be an average value across all the SagaChain shards, and provide average service in terms of transaction queue placement and resulting latency.

Line 7:

extraPerGU is the fee in SagaCoin that the user is willing to pay above the feePerGU for priority placement in the transaction queue and resulting latency.

Line 8:

The return statement must return the dictionary instance at completion of the __hdr() function. Failure to do so with terminate the transaction with an error.

Lines 9 – 13 define the transaction which is the __body() function

Line 9:

Line 9 assigns a new public key associated with a signing key previous generated by the new owner for the new account. This is a convenience. The string may also be directly entered in the following message. Additionally, similar to above the doimport()/CMIImport() pair may be used to read the string from a module.

Line 10:

factoryobj = ClsObjVar(CMIConst.SPAccountFactory)

An object reference to the SPAccountFactory object is created. ClsObjVar(<LOID>) creates a reference to an object managed by the CMI. CMIConst.SPAccountFactory is predefined LOID object as part of the SagaPython execution environment. The list of global LOID objects is provided in the reference section.

factoryobj is an object reference that may be used to send messages to, as the result of executing line 9.

Line 11:

newacct = factoryobj.FactoryNew('200.00', "acctpk")[0]

The FactoryNew message is sent to the SPAccountFactory referenced by the ClsObjVar object reference, factoryobj. The use of the python dot notation syntax provides the look and feel of a standard python method reference. SagaPython translates this internally to a CMI message call to the CMI object. The CMI object is read from the database, and the method associated with the FactoryNew message is executed.

The FactoryNew method implementation takes two arguments while SagaChain is under development before testnet is launched:

- An arbitrary initial amount of SagaCoin
- A public key for the new account

When testnet is launched, SagaCoin may only be distributed by sending from an existing account, or through block rewards. For local testing purposes by developers offline, a version of the FactoryNew message and the associated SPClassAccount will remain available.

Line 12:

Log("New Account Instance: ", newacct.oid)

Line 12 generates output intended to be included in the results section of a new SagaChain block, on successful execution of the transation. The new LOID is output ascii hex format. For readability extra text is provided the example. As excess text increases the cost in gas units of the transaction and

consumes space in the resulting block, it is expected that the output will be kept to a minimum on the live net.

Line 13:

A successful execution returns the boolean True. Any other returned value is considered a failed transaction.

Account Balance Transfer Example

File: newacct1id.py

```
1 newacct1 = 'f7c0751b9f01f5b9bda1402496bec56302067564e246c105551471ff00000000'
```

File: accounttransfer.py

```
def __hdr():
2    nact1mod = doimport('newacct1id.py')
3    CMIImport('nact1mod', 'nact1mod', '*')

4    hdr = { 'accts': newacct1,
                'seq': 1001,
                    'maxGU': 10,
                    'feePerGU': 1,
                    'extraPerGU': 2
      }
   return hdr

def __body():

5    nact1mod = doimport('newacct1id.py')
6    CMIImport('nact1mod', 'nact1mod', '*')
```

```
7    clsvar = ClsObjVar(CMIConst.SPAccountFactory)

   # acctpk is copied here for clarity, could be in an imported module
8    acct2pk = 'verf0-v3e6nhdorn6nusp5d5j2zfi4uvp6tgrxxg7irkupwfcx3rohw7aa'
9    newacct2loid = clsvar.FactoryNew('0.0', 'acctpk')[0]

   # get object references
10   acct1ref = ClsObjVar(LOID(newacct1))
11  acct2ref = ClsObjVar(newacct2loid)



12   Log("acct1 balance: ", acct1ref.GetBalance())
13   Log("acct2 balance: ", acct2ref.GetBalance())
14   Log("send 10 from acct to acct2")
15   acct1ref.SendTo('10.0', newacct2loid)

   Log(" ")
16   Log("acct1 balance: ", acct1ref.GetBalance())
17   Log("acct2 balance: ", acct2ref.GetBalance())

      return True
```

Command lines

Signing the transaction script:

```
19 python ../sagaclient/spclient.py accounttransfer.py -s newaccount.signing.key -o
signedtransactions/signedaccounttransfer
```

Executing the transaction script:

20 python sptransactionexecutor.py -s newdb
testtransactionscripts/signedtransactions/signedaccounttransfer

21 File: sptransactionlog.txt

acct1 balance: 200.00000000

acct2 balance: 0E-8

send 10 from acct to acct2

acct1 balance: 190.00000000

acct2 balance: 10.00000000

Explanation Line by Line

Line 1:

The ascii hex string from the previous transaction that created a new account object is stored in a file and given a name for convenience for future importing to transaction scripts.

Lines 2 – 4 define the __hdr() section

Line 2-3: imports the file newacct1id.py to import the string and string name for the account object LOID

Line 4: The header dictionary has the accts field set to the imported string name newaccts for the account object LOID. Note: the string could also be directly substituted here. Using names improves clarity for human readability.

Lines 5 – 18 define the __body() section

Line 5-6: import the account object id string name, as above

Line 7: creates an object reference to the System Account Factory

Line 8: is a new public key of a key pair generated with edsig. This could have been imported using the doimport()/CMIImport() functions, but is copied explicitly here for clarity.

Line 9: a new account is created here with a zero balance, and its public key for future transactions

Line 10 - 11: create object references for both objects to be able to send messages to them in the following lines

Line 12 – 14: outputs to the transaction log the current values for the SagaCoin balances in each account. The extra text is for human readability. Transactions on SagaChain would be expected to only output values and leave it up to the clients for presenting human readable data.

Line 15: sends the SendTo message to acct1 instruction it to send the amount passed in the argument to acct2. The following verifications occur before the transfer is allowed:

- Signature verification of the transaction, proving the transaction has the right to decrement the acct1 balance. The SPClassAccount class inherits from SPClassVerify. SPClassVerify's verify() method interrogates the transaction script header to verify that the acct1 LOID is listed, and signed. An error is thrown on failure.
- Balance verification is verified by the SPClassAccount SendTo method. A negative balance will throw an error.

Provided that the above checks pass, the SendTo method sends the Increment message to the acct2 object instance. The Increment() method updates the acct2 balance.

Line 16 – 17: Output the updated balances for acct1 and acct2. This output is in human readable form. A SagaChain transaction would provide values only.

The transaction log shows the resulting output.

Note: For this transaction, only success or failure is needed. The account object states reflect the new balances and can be queried as needed.

New Class Creation and Object Creation

File: classassetexample.py

```
# creates a new class and an object for it

def __hdr():
```

```
  sysacct = doimport('sysaccount.py')
  CMIImport('sysacct', 'sysaccount', '*')

  hdr = { 'accts': SystemAccount,
          'seq': 1001,       # whatever the next sequence number is
                'maxGU': 10,
                'feePerGU': 1,
                'extraPerGU': 2
    }
  return hdr

def __CMIClasses():
  @SagaClass(CMIConst.SPClassObject)
  class ClsAsset:

    SagaFieldTable = [
    ]

    @SagaMethod()
    def __init__(self, name: str, initcount: int):
      self.assetcount = initcount
      self.name = name
      self.callcounter = 0

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

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

    @SagaMethod()
    def GetCount(self) -> int:
      return self.assetcount

    # internal method - example simply counts number of calls
8     def callcount(self):
9       self.callcounter +=1

def __body():

  # The name ClsAsset is only available to the transaction script
  # The objectID must be retrieved if the intent is to use the class
  # log it for user to recover it, could also store it in another object
10  Log("Class ClsAsset LOID: ", ClsAsset.oid)

  # An instance of the new class can be instantiated
11  classvar = ClsObjVar(ClsAsset)
12  objloid = classvar.new(CMIConst.SPSystemAccount, "sparkplugs", 100)[0]
13  Log("ClsAsset Object Instance: ", objloid.oid)

14  objvar = ClsObjVar(objloid)
```

```
15    objvar.Increment(10)
16    count = objvar.GetCount()
17    Log("Increment Count: ", count)

    return True
```

Command Lines

Signing the transaction script

python ../sagaclient/spclient.py classassetexample.py -s ../systemaccount.signing.key -o signedtransactions/classassetexample

Execution the transaction script

python sptransactionexecutor.py -s newdb testtransactionscripts/signedtransactions/classassetexample

Transaction Log

Class ClsAsset LOID: abcdefeb0000000000000000000000000000000000000000000000000000000101

ClsAsset Object Instance:
abcdefeb0000000000000000000000000000000000000000000000000000000102

Increment Count: 110

Explanation Line by Line

The header section follows the same model as the previous examples.

Line 1: The __CMIClasses() transaction script section defines new CMI class objects that will be added to the object state database. The syntax for defining a class is similar to defining a Python class, with important differences.

Note: Although SagaPython uses Python syntax, compiler and bytecode interpreter, the classes are not Python classes.

Line 2: @SagaClass(CMIConst.SPClassObject)

A SagaPython class starts with the @SagaClass decorator. The parameters to this are:

- a comma delimited list of LOIDs of base classes
- a account owner field which defaults to SPSystemAccount
- a metaclass field which defaults to SPMetaClass
- a metaclass arguments field which defaults to empty

In this case, line 2 means that the new class inherits from CMIConst.SPClassObject. SPClassObject is a predefined global. It is the root class for all classes. Its value is the following:

SPClassAccount = LOID(bytes.hex(bytes.fromhex('00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07')))

All of the predefined globals may be found in the file sagapythonglobals.py.

Line 3: This defines the class name. Although this is the "class" keyword for Python, it should not inherit from any other Python classes, or specify a Python metaclass. The class name is available only for the duration of the transaction script. As will be shown below, the LOID of the new class must be logged output for future use.

Line 4: The SagaFieldTable defines the fields that are externally visible for instances of the new class. Field definitions will be shown in later transaction scripts. The new class will not have any externally visible fields.

Line 5: @SagaMethod() decorator defines a CMI method for the class. CMI methods are visible externally for instances of the new class. Internal methods do not have the decorator, as is shown farther down.

Line 6: The __init__ method is a SagaPython method, not a Python method. It is called by the CMI when a new CMI object of the new class is created by sending a "new" message, as is shown farther down. Although it technically can be called after the object has been instantiated, the usual use-case is for it to be called by the "new" message one time.

Line 7: The Increment(self, inc: int) method defines a regular method for the new class, which will be called by the CMI when the "Increment" message is sent to an object reference for an instance of the new class. The type syntax, inc: int, follows the Python type syntax notation. Unlike Python, SagaPython implements runtime type checking on passed in arguments as a security measure. The list of argument types is restricted. The list is provided in a reference section. (tbd)

Line 8: The callcount method is known as an "internal" method. It is only visible and may be called under the following rules:

- The method is called from within the code of an externally visible CMI method, such as the ones defined above it.
- The method may not be called from a superclass or subclass method of this CMI class.
- The method must be called with the self variable that is passed in to a CMI method.

Line 9: Attributes that are not defined in the SagaPythonField table (line 4) are internal variables. Similar to line 8 above, they are only visible from within execution of CMI methods of the class. They operate similar to Python attributes such that assigning a value creates the attribute, which is then stored in the object state database.

Lines 10 – 17 define the __body section

Line 10: Writes the LOID for the new class to the output log. The variable name ClsAsset is defined only for the execution of this transaction. Future reference to the class must use the LOID.

Line 11: An object reference is needed to send a message to the new class to create a new object. The ClsObjVar() functions passed in variable ClsAsset creates an object reference to the new class.

Line 12: A new message sent to a class object reference creates and returns a new object instance of the class. A new message sent to a class object always takes at least one argument, the account owner for the new object. The account owner is usually the first account in the header section, and that has signed the transaction. In this case, SPSystemAccount. The account owner parameter may be to an account other than the transaction owner's account, provided that the account and signature are provided in the transaction.

Line 13: As with the new class LOID, the new object instance LOID must be logged for future references to the object.

Line 14: To send a message to the newly created object instance of the new class, a reference to the new object is created.

Line 15 - 17: Demonstrate: calling a method, Increment, with a value; a second method, GetCount, to return the value; and logging the value as resulting output.

The transaction log output is shown above with the values from the execution of the transaction script.

Comment:

After the completion of the execution of this transaction script, the object state database has been updated with a new class with associated code, and a new object instance of the class. New objects of this class may be instantiated in future transaction, and new classes may be defined that inherit from this class.

As SagaPython support multiple inheritance, the base class list in @SagaClass(…) of a new class could include the LOID for this new class, as well as others.

Ancestor Call Operator (_cmi())

This transaction script demonstrates using the _cmi() operator, which functions similar to the Python super() function. The super() method may not be used for CMI ancestor calling because the ancestor class for ClsObjVar is the Python object class, not the CMI ancestors, similarly for the self class in a CMI method. However the following syntax will call the CMI ancestor:

self._cmi().<method>()/<fieldname>

or if a ClsObjVar object reference:

ClsObjVar(<LOID>)._cmi().<method>()/<fieldname>

File: clsassetandassetloid.py

```
1 ClsAsset = 'abcdefeb0000000000000000000000000000000000000000000000000000000103'

2 assetobject = 'abcdefeb0000000000000000000000000000000000000000000000000000000104'
```

File: cmicalltest-createasset-explicitancestorcall.py

```
def __hdr():
   sysacct = doimport('sysaccount.py')
   CMIImport('sysacct', 'sysaccount', '*')


   hdr = { 'accts': SystemAccount,
```

```
            'seq': 1004,       # whatever the next sequence number is
                    'maxGU': 10,
                    'feePerGU': 1,
                    'extraPerGU': 2
        }
    return hdr

def __CMIClasses():
3    assetmod = doimport("clsassetandassetloid.py")
4   CMIImport('assetmod', 'assetmod', '*')

 5   @SagaClass(LOID(ClsAsset))    # no signature required for creating global classes
 6   class ClsAssetSubclass:

        SagaFieldTable=[]       # no user visible fields

        @SagaMethod()
        def __init__(self, name: str, initcount: int):
7           self._cmi().__init__("horns", 55)
            self.assetcount = initcount # internal instance variable
            self.name = name
            self.callcounter = 0

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

        @SagaMethod()
```

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

    @SagaMethod()
    def GetCount(self):
        return self.assetcount

    @SagaMethod()
    def GetName(self):
        return self.name

    # internal method - example simply counts number of calls
    def callcount(self):
        self.callcounter +=1

def __body():
8   assetmod = doimport("clsassetandassetloid.py")
9   CMIImport('assetmod', 'assetmod', '*')

   Log("Class ClsAssetSubclass LOID: ", ClsAssetSubclass.oid)

10  objvar = ClsObjVar(ClsObjVar(ClsAssetSubclass).new(CMIConst.SPSystemAccount, "wipers", 25)[0])

11  Log("ClsAssetSubclass instance: name: ", objvar.GetName())
12  Log("ClsAssetSubclass instance: count: ", objvar.GetCount())
```

```
13    Log("ClsAsset ancestor: count: ", objvar._cmi(ClsAsset).GetCount())

14    prevassetobj = ClsObjVar(LOID(assetobject))

15    Log("ClsAsset instance: count: ", prevassetobj.GetCount())

   try:
      # The ancestor ClsAsset does not have a GetName() method. The _cmi() operator will bypass
      # the ClsAssetSubclass GetName() method and this will raise an error: Unknown method call as
expected
 16     Log("Fail: ", objvar._cmi(ClsAsset).GetName())
   except:
      pass

   return True
```

File: sptransactionlog-cmiexample.text

```
17 Class ClsAssetSubclass LOID:
abcdefeb000000000000000000000000000000000000000000000000000010d
18 ClsAssetSubclass instance: name:  wipers
19 ClsAssetSubclass instance: count:  25
20 ClsAsset ancestor: count:  55
21 ClsAsset instance: count:  110
```

Explanation Line by Line

Lines 1 – 2: The import module provides variable names for the LOIDs of the previously created ClsAsset and an instance of that class.

Lines 3 – 4: Import the module into the _CMClasses() section and makes them visible.

Line 5: Uses the ClsAsset variable for the LOID ascii hex string, first converting it to an LOID, then using the @SagaClass decorator to set it as the base class for the new ClsAssetSubclass class being created.

Line 6: This names the new ClsAssetSubclass for the transaction execution. The same class naming behavior as previous examples.

Line 7: self._cmi().__init__("horns", 55)

This line demonstrates the use of the _cmi() operator to call the ancestor method of the class of the current object. In this case, the ancestor __init__ is called with a different name and different amount arguments from the values passed in from the body section of the transaction script, shown below.

Lines 8 – 9: Import the module with the variable names into the _body() section to make them visible.

Lines 10 – 12: Create a new instance of the ClsAssetSubclass, and output the name and amount values. This demonstrates the override of the GetCount() method, outputting value 25.

Line 13: Uses the _cmi() operator to bypass the current class of the object, and call the ancestor ClsAsset method GetCount() that was overridden by the ClsAssetSubclass GetCount() method, outputting the value 55. This matches the value 55 that was passed to the ancestor in line 7.

Lines 14 – 15: These lines demonstrate retrieving a previously created object instance of ClsAsset, and calling the GetCount() method of ClsAsset. This object shares the same ClsAsset as that of ClsAssetSubclass, but is otherwise independent, as is expected.

Line 16: Demonstrates that the _cmi() operator bypasses the GetName() method of ClsAssetSubclass, and that it is not implemented in the ancestor ClsAsset. It further demonstrates explicitly naming the ancestor class to inherit from. Intervening classes are bypassed. This line throws an error as is expected.

Line 17: Captures the new class, ClsAssetSubclass, LOID for future use. Note that in this case the new object instance of ClsAssetSubclass LOID is not output and would be lost. As there are no remaining references to the object at the completion of the transaction, it will be garbage collected.

Lines 18 – 21: Provide the expected outputs for the various values.

Adding An Importable Module To The Object State Database

The SagaChain environment differs from a traditional Python application execution environment such that importing modules for use by CMI classes can not use the Python “import” syntax directly. Python “import” syntax attempts to locate import modules by name using various search paths. These modules are commonly available in the local file system or through a URL from a webserver or similar.

All of the CMIImport() commands shown in the previous transaction scripts import the specified module only within the context of the, then executing, transaction. At the completion of the transaction, the entire execution environment, including any imported modules, is deleted.

When retrieving an object from the object state database, and the associated class tree, importing modules from the object state database is supported using SPClassModule instances. The following demonstrates creating the SPClassModule instance, and the next transaction script demonstrates creating a class that imports the module.

1 File testimportfile:

```
import_x = 10

import_y = "yahoo"
```

2 ld = LOID(bytes.hex(bytes.fromhex("dead beef")))

```
# Creating a ClassModule instance that includes a module import source file


def __hdr():
   SystemAccount = 'abcdefeb000000000000000000000000000000000000000000000000000000000000'
   hdr = { 'accts': SystemAccount,
            'seq': 1000,       # whatever the next sequence number is
                  'maxGU': 10,
                  'feePerGU': 1,
```

```
                    'extraPerGU': 2
        }
    return hdr

def __body():

    # module source -- will be preprocessed on client side, including manually for test
 3  impsrc = doimport('testimportfile.py')

    Log("SPClassModule LOID: ", CMIConst.SPClassModule)

    # A classv variable for SPClassModule
4    classvar = ClsObjVar(CMIConst.SPClassModule)

    # create a new module object instance
5    objloid = classvar.new(CMIConst.SPSystemAccount, impsrc)[0]

    Log("SPClassModule instance LOID: ", objloid.oid)

6    objvar = ClsObjVar(objloid)

7    Log("SPClassModule instance module src: ", objvar.pym)

    # import the module from the ClassModule instance here and verify variables exist
 try:
 8      Log("before: testmod: ", testmod)
 9     Log("before: import_x: ", print(testmod.import_x))
 10      Log("before: import_s: ", print(testmod.import_y))
    except:
```

```
11    Log("before: no imported_x or imported_s variables")

12   CMIImport(objloid, "testmod")
   CMIImport('impsrc', "localtestmod")

13   Log("after: testmod: ", testmod)
14  Log("after: testmod.import_x: ", testmod.import_x)
15   Log("after: testmod.import_s: ", testmod.import_y)
16  Log("after: testmod.ld: ", testmod.ld)
17   Log("after: testmod.ld.OIDstr()", testmod.ld.OIDstr())

   Log("after: localtestmod: ", localtestmod)
   Log("after: localtestmod.import_x: ", localtestmod.import_x)
   Log("after: localtestmod.import_s: ", localtestmod.import_y)
   Log("after: localtestmod.ld: ", localtestmod.ld)
   Log("after: localtestmod.ld.OIDstr()", localtestmod.ld.OIDstr())

   return True
```

File: sptransactionlog-moduleimporttext.txt

SPClassModule LOID: <CMILOID.LOID object at 0x7f188d71bd90>

SPClassModule instance LOID:
abcdefeb000000000000000000000000000000000000000000000000000000112

18 SPClassModule instance module src:
UEsDBBQAAAAAAPtbgVVeSNAgUQAAAFEAAAARAAAAdGVzdGltcG9ydGZpbGUucHlpbXBvcnRfeCA9IDEw
CmltcG9ydF95ID0gInlhaG9vIgpsZCA9IExPSUQoYnl0ZXMuaGV4KGJ5dGVzLmZyb21oZXgoImRlYWQgYmVl
ZiIpKSlQSwECFAMUAAAAAAD7W4FVXkjQIFEAAABRAAAAEQAAAAAAAAAAAAAApIEAAAAAdGVzdGltcG9
ydGZpbGUucHlQSwUGAAAAAAEAAQA/AAAAgAAAAAAA

before: no imported_x or imported_s variables

after: testmod: <module 'modlib/abcdefeb0000000000000000000000000000000000000000000000000000000112' (modlib/abcdefeb0000000000000000000000000000000000000000000000000000000112)>

after: testmod.import_x: 10

after: testmod.import_s: yahoo

after: testmod.ld: <CMILOID.LOID object at 0x7f188c10bc50>

after: testmod.ld.OIDstr() deadbeef

after: localtestmod: <module 'modlib/impsrc' (modlib/impsrc)>

after: localtestmod.import_x: 10

after: localtestmod.import_s: yahoo

after: localtestmod.ld: <CMILOID.LOID object at 0x7f188c076690>

after: localtestmod.ld.OIDstr() deadbeef

Explanation Line by Line

Line 1: Defines a new import file with various variables that will be stored by a SPClassModule instance for later importing.

Line 2: Demonstrates that the CMI environment for SagaPython is provided automatically.

Line 3: Imports the module into the transaction script such that it may be passed to a new instance of SPClassModule.

Line 4: Creates an object reference to SPClassModule. SPClassModule is a global SagaPython foundation class.

Line 5: Creates a new SPClassModule instance passing it the module that was imported into the transaction script with doimport() above.

Lines 6 – 7: Create an object reference to the new SPClassModule instance and outputs the stored python module to the log. This is for human readability, would not be part of the a transaction on SagaChain.

Lines 8 – 11: Demonstrates the module that was previously stored in the SPClassModule instance has not been imported yet. Attempting to access the variables in the namespace will throw an error as the namespace does not exist yet.

Lines 12 – 17: Demonstrates the available of the module namespace after the import at line 12.

Line 18: Outputs the module import text that is stored in the SPClassModule instance. The source is encoded as printable ascii.

Using An Import Module From the Database

The above transaction script created a SPClassModuleInstance with a module. This next transaction script uses this instance to import the module to a new class such that an instance of the class will have the module imported and in its namespace by the CMI environment.

```
def __hdr():
   SystemAccount = 'abcdefeb0000000000000000000000000000000000000000000000000000000000'
   hdr = { 'accts': SystemAccount,
            'seq': 1000,
                  'maxGU': 10,
                  'feePerGU': 1,
                  'extraPerGU': 2
     }
   return hdr

def __CMIClasses():
```

```
1   modobjLOID =
LOID(b'abcdefeb0000000000000000000000000000000000000000000000000000000112')
```

```
2  @CMIImportDec(modobjLOID, "testmod")
3  @SagaClass(CMIConst.SPClassObject)    # no signature required for creating global classes
 class ClsImportTest:

    SagaFieldTable=[]       # no user visible fields

    @SagaMethod()
    def __init__(self, name: str, initcount: int):
       self.assetcount = initcount # internal instance variable
       self.name = name
       self.callcounter = 0

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

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

    @SagaMethod()
    def GetCount(self):
       return self.assetcount
```

```
    @SagaMethod()
    def GetTestmod(self):
4       return testmod.import_x

    # internal method - example simply counts number of calls
    def callcount(self):
        self.callcounter +=1

def __body():

  Log("Class Import Test LOID: ", ClsImportTest.oid)    # log it for user to recover it, could also store it in
another object

  # An instance of the new class can be instantiated
5 classvar = ClsObjVar(ClsImportTest)

6   objloid = classvar.new(CMIConst.SPSystemAccount, "importstuff", 9999)[0]    # initialized with 100
count of asset
  Log("ClsImportTest Object Instance: ", objloid.oid)

  objvar = ClsObjVar(objloid)

7   count = objvar.Increment(10)

8   Log("Increment Count: ", count)

9   Log("testmod test: ", objvar.GetTestmod())

  return True
```

10 File: sptransactionlog-classimporttest.txt

Class Import Test LOID: abcdefeb000000000000000000000000000000000000000000000000000011c

ClsImportTest Object Instance:
abcdefeb000000000000000000000000000000000000000000000000011d

Increment Count: 10009

testmod test: 10

Explanation Line by Line

Line 1: Creates a variable for the previously created SPClassModule instance LOID. For clarity it is directly included. It could also have been doimport()/CMIImport() loaded.

Line 2: @CMIImportDec(modobjLOID, "testmod") decorator defines that the module in the SPClassModule instance, modobjLOID, should be imported into the environment for executing the following class's methods and fields, and given the namespace name, "testmod". Multiple @CMIImportDec decorators may be stacked. They are processed in order from top to bottom.

Line 3: Decorator defines the class the same as previous examples. The @SagaClass() decorator must be the last decorator in the list of decorators before the class name.

Line 4: Demonstrates the availability of the testmod namespace and access to objects in the module, which are imported into the object instance space of the defined class when the method is executed. In this case, the method GetTestmod() will return the value in the testmod module. Imported modules do not maintain state across transaction executions. That is, the module is freshly imported from the SPClassModule instance in the object state database for each new transaction.

Lines 5 – 6: Demonstrate creating a new instance of the new class, ClsImportTest, defined above.

Lines 7 – 8: Demonstrate method execution the same as previous examples.

Line 9: Demonstrates calling the method that contains the access to the object in the imported namespace testmod, via the code in line 4 as defined by the decorator of line 2.

Line 10: The log file outputs the resulting values showing access to an object in the imported module from the SPClassModule instance.

Examples Locations

- The transaction scripts for the above examples may be found in the testtransactions subdirectory of the project source. The examples must be run out of this directory to locate the various import module files currently.

- The spclient.py is located in the sagaclient subdirectory.

- The sptransactionexecutor.py is located in the main project directory.

- Edsig, the key generation tool is located in the main project directory.

Reference

Tools

Edsig

Key generation tool. Generates a signing key (private key), and a verification key (public key). When creating new accounts, the public key is a parameter to the FactoryNew message to create the account. Subsequent messages to objects owned by the account verify the signature in the transaction script signed with the signing key.

Command line

Usage: (ed25519 version 1.5)

edsig generate [STEM]

creates keypair, writes to 'STEM.signing.key' and 'STEM.verifying.key'

default is to 'signing.key' and 'verifying.key'

Spclient.py

spclient.py is the client side tool that takes an unsigned transaction script as input and signing keys and generates a signed transaction script. Spclient.py also replaces the doimport(<filename>) function with the source code of the named file, and assigns the source code as hex ascii encoded data. The output is intended to be ready for submission to SagaChain. Currently this is simulated with the direct call to sptransactionexecutor.py.

Command line

spclient.py - usage:

--help display this help message

--first-operand <filename of transaction script> is required, plus optional switches

-o ARG --outfile ARG processes transaction script output file

-s ARG --sig ARG signing key

Sptransactionexecutor.py

Sptransactionexecutor.py is the SagaChain node side application that executes the transaction script. The SagaChain nodes will use this application to execute transactions included in each block, and update the object state database as part of the consensus protocol. Currently this is simulated with a direct call to the application.

Command line

sptransactionexecutor.py - usage:

--help display this help message

--first-operand <filename of transaction script> is required, plus optional switches

--hdrwrapper Name of the header wrapper file to compile - default sagahdrwrapper.py

--bodywrapper Name of the body wrapper file to compile - default sagabodywrapper.py

--classeswrapper Name of the classes wrapper file to compile - default sagaclasseswrapper.py

--tailwrapper Name of the tail wrapper file to compile - default sagatailwrapper.py

-d ARG --decquant ARG Decimal quantization. Must be form ".00001" - default is ".00000001"

-s ARG --instatedb ARG Input leveldb object state database filename - default is sagapython.db

-r ARG --readstatedb ARG output leveldb database filename for all objects that were read during transaction execution - default is sagapythonreadout.db

-w ARG --writestatedb ARG output leveldb database filename for all objects that were created or modified during transaction execution - default is sagapythonwriteout.db

-l ARG --transactionlog ARG log file of transaction results

Foundation Classes

SagaPython includes a set of builtin foundation classes and object instances that “bootstrap” the runtime environment. These are created one time and stored in the object state database. The sptransactionexecutor.py simulates this by checking for the existing of these classes and objects each

time the application is launched. SagaChain will contain a genesis block with these objects. The classes and object instances are:

Foundation Classes

SPClassAccount

SPMetaClass

SPClassAccount

SPClassSystemAccount

SPClassModule

SPClassAccountFactory

SPClassVerify

The source code for these classes may be found in the product/ directory by filename. Refer to the baseobjects.py for the order of class and object creation.

Foundation Object Instances

SystemAccount

AccountFactory

TransactionObject

References

SPClassObject

SPClassObject is the root class for all classes. The inheritance for all classes eventually reaches SPClassObject. Unimplemented methods and fields will throw an error if the resolution reaches SPClassObject.

Implements fields:

- SPClassObject.CMIConst.CMIType the class type of the object instance.
- SPClassObject.CMIConst.CMIOwner the account owner of the object instance. All objects are owned by one account at a time. Accounts own themselves.

SPClassAccountFactory

SPClassAccountFactory follows the “factory design” pattern. It creates new accounts by creating new account object instances.

Implements methods:

- FactoryNew takes the verification key (i.e. public key) for signature verification. Initial design takes a Decimal amount of initial SagaCoin. SagaChain will only allow creation with the public key, and the sendto() method of a SPClassAccount object to add SagaCoin to the account. For development purposes, accounts may be created with initial balances in transaction script.

SPClassAccount

SPClassAcount manages the SagaCoin balance of the account, the state root for all objects contained by the account, and the verification key (i.e. public key). Additionally, account object instances of SPClassAccount provide a base for new LOIDs to be derived when creating new class and objects owned by the account object instance.

The base for every object consists of the high order 224 bits of the LOID, concatenated with 0’s for the remaining 32 bits. New classes and objects are created with an incrementing value stored in the account object state. This ensures that new classes and objects may be created in parallel on different shards without concern of LOID value collisions.

Implements Methods

Increment(amount) – takes a SagaCoin amount either as a decimal string, or a Decimal object. Increments the SagaCoin balance

Decrement(amount) – takes a SagaCoin amount either as a decimal string, or a Decimal object. Decrement throws an error if the balance would go negative, and does not decrement the balance in that case. SPClassAccount inherits from SPClassVerify and calls the verify() method in the Decrement() method to verify that the transaction script contains a matching signature for public key of the account.

SendTo(amount, account) - takes an amount of SagaCoin and a destination account to send the balance to, from the account calling the SendTo method. Internally, SendTo calls Decrement and will throw an error if the resulting balance would be negative, or verification fails. The Increment method does not perform signature verification. An account will receive SagaCoin from any source.

GetBalance() – returns the current balance of SagaCoin.

SPClassModule

SPClassModule objects store Python modules that may be imported by either a CMI class using the @CMIIImportDec() decorator or using the CMIIImport() function in a section of transaction script when an instance of SPClassModule is passed to the CMIIImport(<loid>) function.

Implements Methods

New - the new message takes the import module in encoded ascii hex format as a string or bytes type object.

SPClassVerify

SPClassVerify provides signature verification methods which may be called by methods of classes that require caller signature verification. The SPClassAccount Decrement() method is an example. Classes that need this feature may inherit from SPClassVerify.

Implements Methods

Verify – verifies that the current object's account owner's verification key matches the signature provided in the transaction script.

VerifyUser – verifies the signature as verify above. It further requires that the verification key match the first account in the list in the transaction script. By definition, the first account in the accounts list is the user submitting the transaction script.

Foundation Object Instances

The LOIDs for these objects are published and well known. Their definitions may be found in the sagapythonglobals.py source file.

SystemAccount

SystemAccount provides the bootstrap account. All objects must be owned by accounts. The SystemAccount serves the role of the initial account for all foundation classes and objects. SystemAccount is initialized with the systemaccount.verification.key. The systemaccount.signing.key is published such that transaction scripts that send messages to objects owned by SystemAccount are signed with the systemaccount.signing.key.

AccountFactory

AccountFactory provides the bootstrap object to create new accounts. It is owned by SystemAccount. Transaction scripts use the systemaccount.signing.key and list the SystemAccount in the header section accts list.

TransactionObject

TransactionObject is a unique global object. It is available to all objects during transaction execution. It provides access to the transaction script contents.

Implements Fields

Header – the source code of the header section

Tail – the source code of the tail section which was added by the spclient.py

Classlist – the list of new class LOIDs defined in the _CMIClasses() section

Accts - the list of account LOIDs for matching signatures added by the spclient.py

Seq - the current executing transaction sequence number of the user account submitting the transaction

maxGU – the maximum gas units this transaction may consume. A gas unit is roughly equivalent to the execution of a byte code by the Python interpreter. However, various features such as message resolution and dispatch consume a single gas unit independent of the number of byte codes executed.

feePerGU – the fee in SagaCoin the user is willing to pay per gas unit for the execution of the transaction

extraPerGU – an extra fee on a per gas unit basis that the user is willing to pay to move the transaction towards the front of the transaction queue.

Hash – the hash of the signed transaction added by the spclient.py

Sig – the signature list of the hash by the signing keys added by the spclient.py

Supported Restricted Argument Types

CMI Decorators

@SagaClass(*baselist, owner=sagapythonglobals.CMIConst.SPSystemAccount, metaclass=sagapythonglobals.CMIConst.SPMetaClass, metaclassargs=(), classfinal=False)

Baselist - SagaClass takes a list of base classes the new class inherits from. SagaPython supports multiple inheritance using the C3 algorithm from standard Python.

Owner – defaults to the SystemAccount object. Classes may be owned by any account object. Various restrictions on access to the class may be implemented based on the class object account owner.

Metaclass – defaults to SPMetaClass. The CMI is a first-class object model. Every object has a class. The class of SPMetaClass is SPMetaMetaClass. (Technically the class of SPMetaMetaClass is SPMetaClass to terminate the regression). The SPMetaClass creates classes. A class that inherits from SPMetaClass therefore could provide additional functionality when creating a new class.

Note: inheriting from SPMetaClass has not been investigate at this time.

Metaclassargs – default is empty. The argument list would be sent to subclasses of SPMetaClass to provide additional functionality when creating a new class.

Classfinal – defaults to false. Setting of this flag creates a class that may be instantiated but not inherited from. This feature may be useful to prevent overriding of methods and fields for a class, ensuring that the functionality for instances of the class is inviolate.

@SagaMethod(*args, **kwargs)

Specifics that the following method definition is a CMI method.

The args list and kwargs list provide flags for access control to the method. (tbd)

@CMIImportDec (loid: LOID | str | bytes | bytearray, localname: str, *args)

Loid is either an LOID object specifying the SPClassModule instance to import the module from, or it is the object module itself in hex ascii format.

Localname is the name for the module to have in the namespace of the transaction or class methods

Args is a list of any of the symbol names in the module to provide direct access to without the qualifying localname syntax. The '*' indicates importing all symbols.

CMIIImportDec() provides similar semantic so the Python syntax: from <filename> import <symbol name> as <name>

LevelDB Class Manager Infrastructure Object Structure

The database is a persistent store. This means that it may be copied, backed up, and shared during development of new classes and applying transaction scripts. If the -s flag references a non-existent database, a new database will be created from scratch with new copies of the foundation classes and object.

- To delete the leveldb databases - rm -rf <database name>.

An object in the LevelDB database consists of the following JSON encoding:

- A list of entries representing the state of the object, keyed by the LOID
- Each entry in the list consists of a list of 2 entries
- The first entry is the external facing methods and fields defined as a SagaPython object for the specific ancestor class of the object.
- The second entry is the internal python objects defined for the specific ancestor class of the object

*Base64 encoding is used to guarantee that all data is string based. This is necessary because the individual methods defined for a class are stored in source form, not byte code, (and are compiled on demand).

- The external data is JSON encoded and base64 encoded.
- The internal data is python pickled and base64 encoded.

As a result, the leveldb objects are opaque lists of lists of base64 encoded strings. Tools that decode base64 strings can convert the external data back to JSON text.

The internal python pickled data may or may not unpickle directly. Some python objects may be dependent on the module environment of the CMI class object at runtime.

A LevelDB tool like leveldb-tools written in go can be used to dump the leveldb database. Refer to GitHub - rchunping/leveldb-tools: import, export or repair leveldb. Will require installing the go runtime environment. The contents of the database will be base64 encoded lists as described above.

Known Issues

SagaPython is alpha code. Some amount of continuous integration tests have been developed, while full code coverage has not been established. It is relatively stable. The following are known issues:

- SagaPython message dispatch and field resolution does not perform argument type checking. Passing incompatible types may have undefined results

- SagaPython uses a significant amount of validation checking throughout and throws Python exceptions on most failures. Internally it uses nested "exec()" calls which results in lengthy and somewhat confusing error messages. Recommend looking for the text specifying the original location of the exception in the error message.

- The CMI is intended to enforce class inheritance rules, method and field access flags, and account boundaries for message dispatch. Most of the rules are yet to be implemented. It is possible to create sets of transactions what will function correctly in the development environment that will not work as-is on SagaChain, when these rules will be enforced.

- SagaPython has not been optimized for performance. In particular method and field resolution iterates over the class inheritance ancestry. This creates the effect that the deeper the inheritance tree the more effort for method and field resoltuion. Future versions will address implement optimization to eliminate the resolution search.

- SagaPython reads the CMI class's method source code from the object state database for each separate execution of the sptransactionexecutor.py. The code is dynamically compiled and cached internally for the length of the execution. Future versions will cache compiled versions of the source code locally for optimization. This will function similarly to standard Python bytecode caching, managed as CMI class code.

Note: All SagaPython methods are stored as (base64 encoded) source code in the object state database to allow for direct inspection.