Smart Contracts

By Marin Peko and Archie Chaudhury

Introduction

Contracts in A1 programming language are similar to classes in Python programming language since amongst Python's virtues is a relatively shallow learning curve and, thus, it should be easy to start writing smart contracts in A1 programming language. Each contract can contain declarations of state variables, functions, struct types etc.

State variables and other objects

State variables are variables that are permanently stored in contract storage. Valid state variable types match all the types supported by the Python programming language. A1 should also support structures/objects that essentially have the ability to be dynamically acted upon within the framework of the larger contract.

Example:

```
contract ExampleStorage:
    X = 13
    store(x)      ## state variable, stored in the permanent storage
of the contract

    delete(x) #delete from storage
    ...
```

Functions

Functions are the executable units of code defined inside the contract.

Example:

```
contract ExampleStorage:
    def pay():              ## function
        ## function body
```

Modularity

A1 should have a collection of standard libraries that developers can leverage to easily create more complex smart contracts. A developer wanting to create an application that features decentralized governance should be able to do so in as little as 5 lines of code by importing the associated libraries and then simply extending them.

Example:

```
import governance

contract Voting:
     start_vote(‘token”,”vote 1”,3,self.storage)
#Voting period with governance token “token” and 3 options
     end_vote()
     return_results(send_message)
```

Modules should allow users of A1 to easily create powerful dApps without having to focus on maintenance.

Type Checking and Declarations

A1 should have some degree of type checking: developers should be able to declare types and check them through an unique declaration. Note that this case only applies if the developer intends for the default type to be not used for a particular function. This dynamic type check is performed during compilation. This should be enabled for int32, int64, int128, int256, and uint32, uint64, uint128, uint256. It should also be enabled for strings, floats, and other elementary data types.

```
contract ExampleStorage:
    stored_value: int ##Check during compilation to see if value is
an int
    def random(x: string):: int #checks that input is string and
return value is int
```

Structures, Inheritance, and more

Within a contract, structures (classes) should be able to be defined, and these classes should support basic inheritance throughout the contract. Inheritance, especially between different structures, should be limited somewhat to prevent mass errors (more on this later). Polymorphism (the ability to execute similar instructions for different structures or objects) should also be available.

Example (add this later)

Distributed Computing (Promises, and other things)

Promises can be assertions or declarations within a program that define certain parameters. This can essentially be an easier, inline alternative to formal/knowledge verification for developers who want to define how their contract will behave in the context of multiple parties interacting with it. Promises and other distributed computing concepts can be ingrained within messages between parties: a contract should be able to define (and test!) interactions between different parties. This can be used for basic security principles for on-chain contracts.

Messages between users/parties interacting with a contract (multi-party smart contract computation) can enable basic promises. For example, in a smart contract defining some sort of bridge from one blockchain to another, a promise can be made that essentially ensures that the user cannot interact with the underlying funds until their deposit has been converted, and if they attempt to do so, their deposit will be automatically burnt. With bridge technology specifically, an automatic assertion should also be able to reference the current version and check signatures through an oracle service, thus ensuring that the deposit of native tokens is valid before initiating a transfer.

Formal/Knowledge Verification

Building on promises, Knowledge Verification allows developers to make certain assertions about their programs. These assertions can be formal (ie, about code itself, such as a constant in a while loop or a mathematical comparison between two variables) or knowledge-based (promises: ensure that an account, for example, deposited the correct amount of tokens in some smart contract or that they completed some action). These actions can be declared within the code: we borrow from both traditional programming literature and the E Programming Language:

```
import token
contract NewToken:
    let owner: address = msg.sender
```

```
    def create(name: str, balance: num):
        token.create(owner, name, balance)

    def transfer(new_owner: address, amount: num):
        if owner != new_owner:
            assert(owner[“balance”] > amount) #Not the best example,
but you get the general idea
            owner["balance"] -= amount
            new_owner["balance"] += amount
```

Promises can be used for specific use-cases: namely, they can help prevent rentrancy attacks and ensure that certain actions are completed before subsequent actions are taken. The promises keyword should be “->”; it should be used to ensure certain actions (such as setting a balance to zero after a withdrawal) happen in the order they are intended. Read the E Rights book for more information: http://wiki.erights.org/wiki/Walnut/Distributed_Computing.

Generalized Tasklist/Things to do in the codebase/Review,Feedback,and random thoughts

(Person taking the lead, person supporting)

Define a common utils library for developers to use within the ecosystem (please note that this is not an A1 library, but rather a general library which contains hashing functions, common data interpretations, ABI definitions, and other basic data structures for the Adamnite Ecocsytem. Documentation on this can be found in the Technical Paper The ETHDEVCORE library is a great reference. (Jonah,Archie).

Develop an implementation/library for the ADVM within the A1 repository. This is for more efficient compilation: this will include an implementation of VM instructions, assembly interpretation, control flow, and some references of state, among other things. While this should follow the original documentation, the ETHEVMASM library is a great reference. (SDMG15, Archie,Jonah)

Develop the core A1 library. This will include the tokenizer, abstract syntax tree, types, parser, and compiler. The syntax (function definitions, keywords, etc) should match Python’s, with smart contract functionality and modularity added. A compiler that compiles A1 source code down to VM code will also be a part of this tasklist. Formal verification/promises will be helpful, although they are not required (Marin, Mike, Ajai).

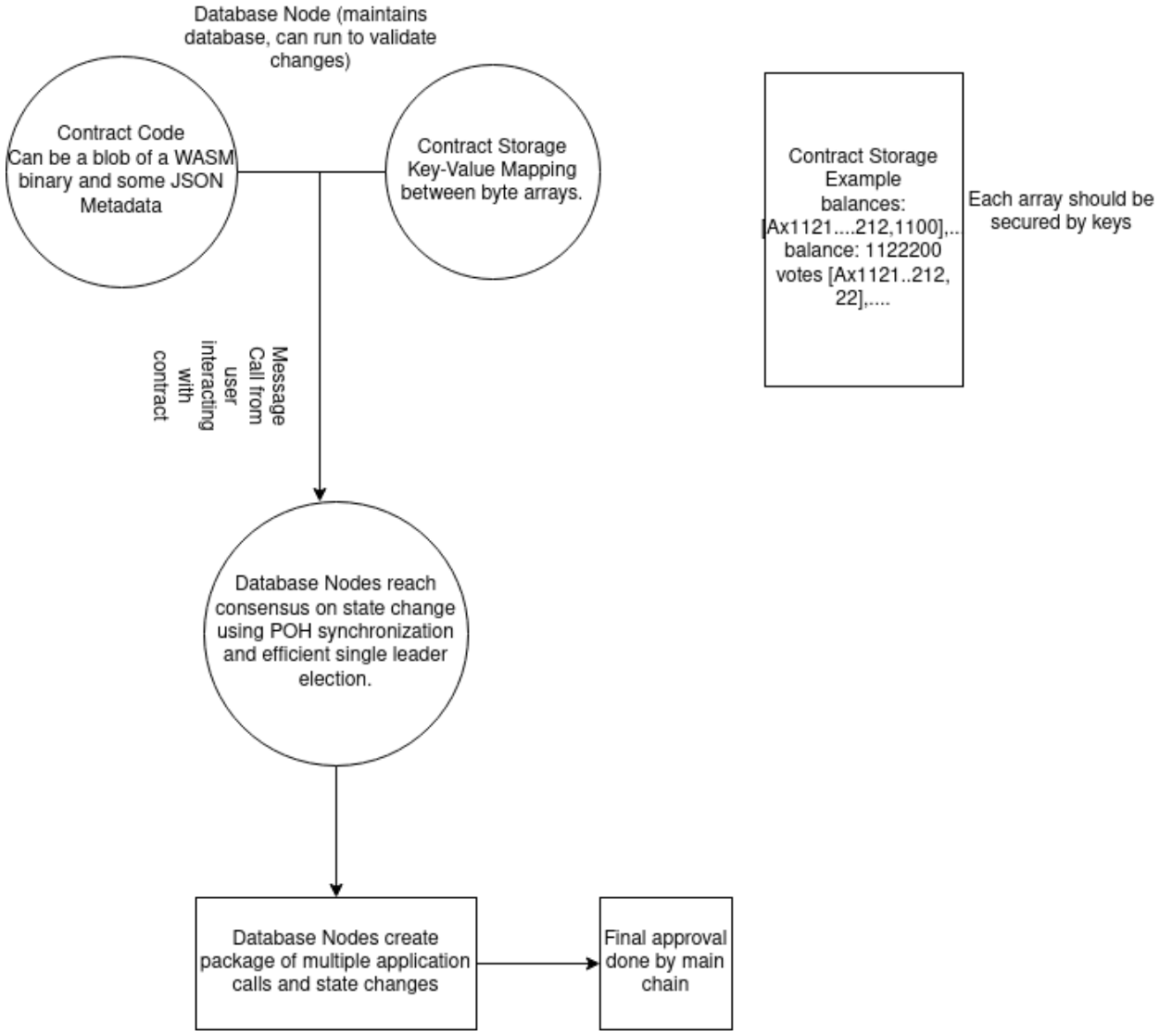
Generalized Feedback for A1 (8/22):

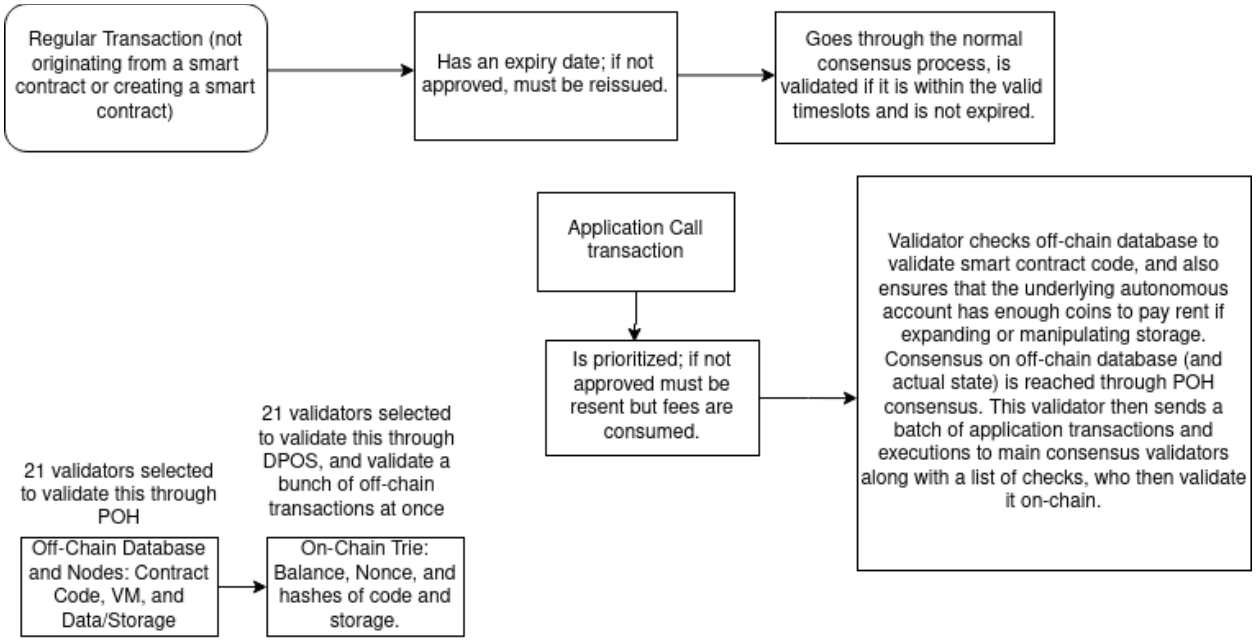
- Add comments before substantial blocks of code (can be done later) to help describe what is happening at different stages. This will especially be helpful as the project becomes larger and larger, and starts to have more contributors.
- Add support for accessing and changing storage, at least on the token side (this will be more clear once clearly define what storage for contracts will actually look like).
- Add support for the following optional type declarations: uint32, int32, uint64, int64, and so on and so forth till 1028. Add support for address (just in tokens, actual support will again come with development of compiler)
- Within expression/abstract syntax tree, add support for JSON Converters (might actually be really helpful for generating JSON metadata for uploading to the off chain database). This can certainly build upon the JSON support in the utils directory.
- Add modularity in the form of packages: developers should be able to create general packages and import them. General packages can apply, but more complex ones supporting the creation of voting, DAOs, etc should also be supported.

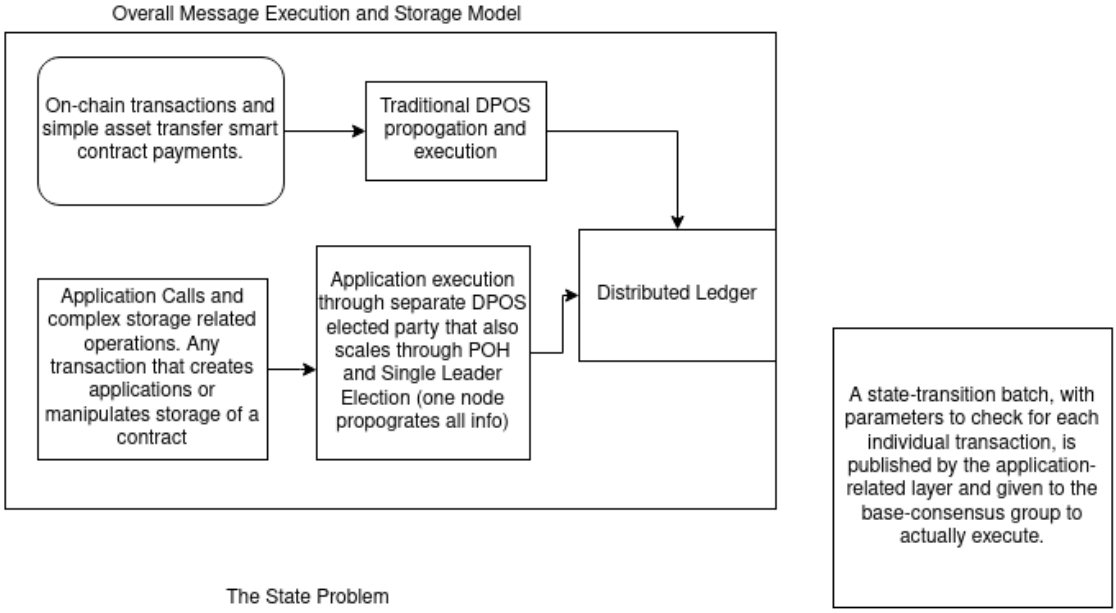
A1 blockchain-specific keywords:

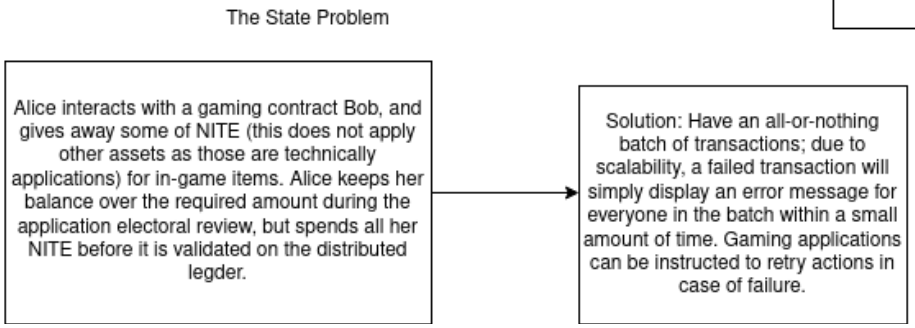
ABI: Defines a contract's functions, names, and other interfaces that an external contract can use to interact with it.
Address: A data type; it defines an average address within Adamnite. It should have the same behavior as num or string: declaring something as an "address" should allow it be used in a certain way.
Block: A class used to get information about the most block. For example, using "block.timestamp" should return the timestamp for the most recent block.
Msg: Already used in examples. Defines a call within Adamnite's formal execution model. Using msg['sender'], for example, points to the address of the caller.
Transaction: Defines a transaction in NITE between two addresses. This is not included within the formal execution model, as it does not require a call to the VM.

Storage and Off-Chain DB model:




Regular Transaction (not originating from a smart contract or creating a smart contract)
Has an expiry date; if not approved, must be reissued.
Goes through the normal consensus process, is validated if it is within the valid timeslots and is not expired.
Application Call transaction
Is prioritized; if not approved must be resent but fees are consumed.
Validator checks off-chain database to validate smart contract code, and also ensures that the underlying autonomous account has enough coins to pay rent if expanding or manipulating storage. Consensus on off-chain database (and actual state) is reached through POH consensus. This validator then sends a batch of application transactions and executions to main consensus validators along with a list of checks, who then validate it on-chain.
21 validators selected to validate this through POH
21 validators selected to validate this through DPOS, and validate a bunch of off-chain transactions at once
Off-Chain Database and Nodes: Contract Code, VM, and Data/Storage
On-Chain Trie: Balance, Nonce, and hashes of code and storage.


Overall Message Execution and Storage Model
On-chain transactions and simple asset transfer smart contract payments.
Traditional DPOS propogation and execution
Application Calls and complex storage related operations. Any transaction that creates applications or manipulates storage of a contract
Application execution through separate DPOS elected party that also scales through POH and Single Leader Election (one node propogrates all info)
Distributed Ledger
A state-transition batch, with parameters to check for each individual transaction, is published by the application-related layer and given to the base-consensus group to actually execute.


The State Problem
Alice interacts with a gaming contract Bob, and gives away some of NITE (this does not apply other assets as those are technically applications) for in-game items. Alice keeps her balance over the required amount during the application electoral review, but spends all her NITE before it is validated on the distributed legder.
Solution: Have an all-or-nothing batch of transactions; due to scalability, a failed transaction will simply display an error message for everyone in the batch within a small amount of time. Gaming applications can be instructed to retry actions in case of failure.