Integrate Burrow EVM into Hyperledger Fabric

Date	2017.11
Authors	Swetha Repakula <srepaku@us.ibm.com> Jay Guo <guojiannan@cn.ibm.com>

Introduction

Ethereum created the Ethereum Virtual Machine (a.k.a EVM), a Turing-complete stack machine, to execute smart contract in EVM-bytecode, which is compiled from high level languages, such as Solidity, Serpent, LLL, Mutan and Viper. From the EVM point of view, once those programs are compiled into EVM-bytecode, original choice of language is irrelevant.

EVM is currently part of Ethereum codebase, which is licensed under LGPL-3.0.
Burrow is a permissioned blockchain framework that has its own EVM implementation that is Apache 2.0 licensed, so that programs written in those smart contract languages can run in a consortium chain leveraging a Hyperledger project such as Sawtooth or Fabric, instead of a public chain such as Ethereum.

Motivation

- Many DApps (Decentralized Applications) written in Solidity or other EVM languages can benefit from being able to run in a permissioned blockchain like Fabric with minimum modification.
- We could improve developer experience of Fabric by allowing developers to write chaincode in Contract-Oriented Languages like Solidity and Viper, especially for those who are already familiar with them. It also helps us to access the large mindshare held by people using Ethereum in the enterprise.
- There are many toolings around these languages to facilitate the development of smart contracts.
- EVM only has a limited set of instructions, which is believed to be more secure than permitting a full range of instructions

Goal

This integration should provide Ethereum developers with similar dev experience. More specifically, a developer should be able to install and invoke a smart contract by sending a transaction to Contract Address via JSON RPC API. Therefore, with minimum modification, an Ethereum DApp could be migrated to Fabric, where it gains benefits of permissioned blockchain, e.g. data privacy & isolation, higher throughput, etc.
Technically, it can be broken down to three layers:

- **EVM ⇔ ledger**
  Burrow EVM accesses underlying data via a set of write/read APIs. We need to implement them to adapt to Fabric ChaincodeStubInterface
- **Peer ⇔ EVM**
  Invocations of an Ethereum smart contract are sent to EVM in the form of transactions. To adapt to Fabric, transactions should be relayed to EVM via Chaincode interface, more specifically, `invoke` API
- **Fabric gRPC API ⇔ Ethereum JSON RPC API**
  Ethereum exposes JSON-RPC APIs for applications to interact with a node. It could be used to install contracts, query state, etc. They are wrapped into Web3.js library that many Node.js apps use. We should support Web3.js tooling by implementing JSON-RPC layer in addition to current gRPC layer.

We should keep this work as least invasive to Fabric core as possible. With the support of pluggable system chaincode, this integration could be implemented as a system chaincode plugin, namely **evmscc**, which keeps Fabric core intact.

Non-goal

- Compilers will not be included into Fabric. So developers need to compile Solidity/Viper using available compilers, e.g. solcjs, and install EVM-bytecode in Fabric.
- In this project, we will ONLY integrate Burrow EVM and JSON RPC API into Fabric, but not Burrow consensus model.

Use cases

TBD

Design Highlights

How to keep up with the evolution of Burrow-EVM?

Currently Burrow EVM is not exposed as a library that we could import in Fabric. Sawtooth copied that piece of code into their codebase. If we could extract it from Burrow into a common library, both Fabric and Sawtooth could benefit from it. Also, Burrow could rely on the Hyperledger community to maintain it.

Conclusion: this work is in progress, see burrow hypermarmot branch
[UPDATE 2.21.18]: Burrow refactor has been merged into their developer branch. The hypermarmot branch has since been deleted.

Should we eliminate the notion of ‘accounts’ and ‘gas’?

In Ethereum, there are two types of account: Externally Owned Accounts (EOAs) and Contract Accounts, both of which have `ether` associated. When an EOA invokes a Contract Account, it purchases `gas` with `ether` at an arbitrary `gas price` (higher gas price is higher bid, which would probably be executed by miners first). These `gas` are used as fuel for EVM to execute actual contract bytecode as a means to protect computation resources of miners from executing

malicious code (infinite loop, etc). However, this mechanism is probably not necessary in the context of permissioned blockchain, also we don't have such semantics for other chaincode languages anyway.
Counter-argument could be that permission is not necessarily binary (allow v.s. forbidden), instead we could leverage gas to achieve various level of privileges. Also, if we do eliminate it from Burrow-EVM, it would make it harder to have common lib.

Conclusion: `Contract Account` in EVM will be conceptually 1-1 mapped to `ChaincodeID` in Fabric, and code associated to `Contract Account` will be stored in ledger. We will discard the notion of `EOAs` (simply mock it?). `Gas` will be retained with a high default value, e.g. 90000 in Sawtooth. More logic can be implemented to make this configurable by, e.g. network operator, contract developer, user with an upper limit. But for MVP, we will stick to hard-coded default.

New Design Decision: Smart contracts that are in evm bytecode will not be managed as other chaincodes in fabric, but rather will be managed by the evmscc.
Sometimes, evm code requires the Externally Owned Account (EOA) semantics, e.g. `msg.sender`. Therefore we will follow the pattern of Ethereum and Burrow, where the account address is computed from sender/creator's public key. In terms of Contract Account, the address will be derived from caller/installer's address plus contract sequence number.
We probably need to add 'export address' functionalities to `cryptogen` util, so that web3 or cli would know the address of a specific user in the network.
In order to be Web3 compatible, we need to have hex contract addresses, and cannot use canonical names. In order to avail of all the features of the Web3 library such as deploying a contract, updating etc, we need to have a slightly different design. Deploying becomes a problem because deploying a smart contract is treated the same way as a transaction which is different from how chaincodes are installed in Fabric. One way to solve this is to have all the smart contracts to be stored in the `evmscc` ledger and allow the `evmscc` to manage the smart contracts. This solves both the problem of generating hex contract addresses for the smart contracts as well as being able to manage smart contracts using web3.
We could possibly incorporate Ethereum Name Service (ENS) as part of this work, so that canonical names of smart contracts could be retained.

How to enable cross-chaincode invocation?

In Burrow EVM, when contract A invokes contract B, it calls `GetAccount` to load the bytecode of B, and execute it in the same vm. However, in Fabric, this type of call is routed through peer via `InvokeChaincode` interface. We could implement the semantic of "retrieving the code of a contract" in Fabric to fulfill `GetAccount`, e.g. translating it to `GetState`, in which case, contract code is stored in the evmscc ledger with key being the address of account. Also, there are `EXTCODESIZE`, `EXTCODECOPY` and `BALANCE` opcodes in Burrow EVM that call `GetAccount` as well (we probably don't care about the last one).

Conclusion: We solve this issue by storing the contract in evmscc ledger, so that GetAccount retrieves requested contract code via GetState. Note that we don't support cross-chaincode invocation among evm smart contracts and Fabric chaincodes. However, we should consider this if there's a demand to call Fabric chaincode from Ethereum smart contract, especially when different types of smart contracts co-exist in the same channel, and need to access same data.

Should we check permissioning when EVM contracts invoke other EVM contracts?

Option 1**:** We have the option to make all EVM contracts in a channel open to each other
Option 2: Cross-invocation has to go out to the peer to validate permissions.

Should we use system chaincode to execute EVM bytecode?

In current design, at least for PoC, we will go for **one-evm-per-channel**, which is used to execute any contract bytecode in the same channel. To actually implement this, we have 2 options:
Option #1, wrap Burrow-EVM, which is now being separated as a library, into a go chaincode (user chaincode).
Option #2, add a new system chaincode, e.g. `evmscc`,

Conclusion: we will go for the second option, mainly because we don't have a mechanism to dispatch txs to a **singleton user chaincode** based on type. In another word, it's hard to implement *"if this tx is targeting ABC chaincode of type EVM, send it to XYZ chaincode"*, however it's to do *"if this tx is targeting XYZ chaincode of type EVM, send it to `evmscc`"*.

How to enforce ACL for JSON RPC requests?

One way to add the web3 JSON RPC support in fabric is to create a JSON RPC server that uses the SDK to communicate with Fabric. However the SDK is used 1-1 with an app. In order to allow multiple apps to talk through the one server/SDK we need to think about access control as not everyone who can connect to the server will be permissioned to talk to Fabric. Currently the JSON RPC requests we receive only have have the callee account address and the caller account address. In order to use the SDK to communicate with Fabric, more information is required to pass along the request.

Option #1: One way is to have a server per channel per org, and leave access to the org controlling the server.e
Option #2: Developers looking to migrate their apps might expect to follow a few steps for the transition. One of those steps could be to set up this server locally for their app and supplying it with the certs for the one app which would follow the 1-1 app to SDK model.

What should the developer experience be for someone migrating an app?

Design Details

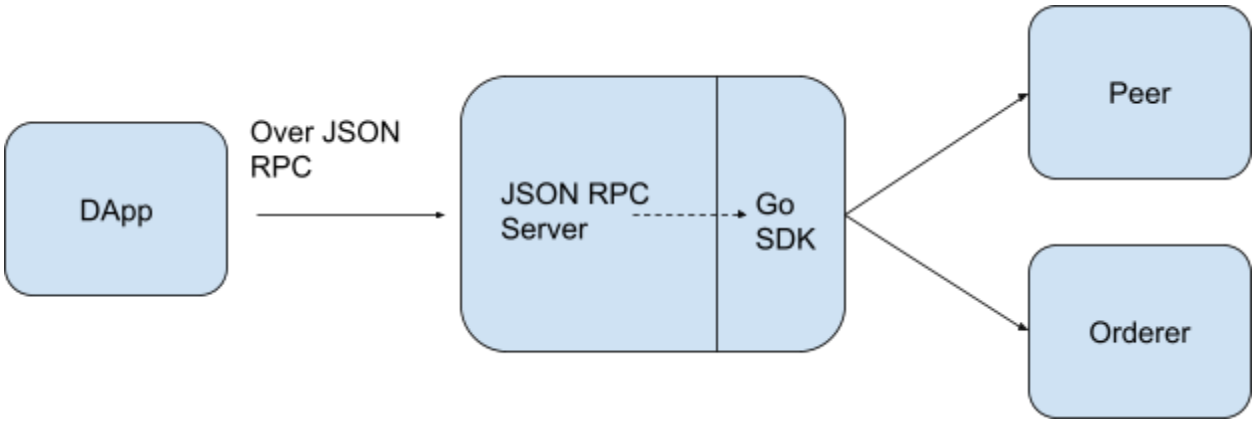
UX

This section describes how users/developers interact with Fabric EVM chaincode

Chaincode deployment

API

Ethereum has produced a JSON RPC API that allows distributed apps (DApps) interact with smart contracts and invoke transactions. One of the libraries that implements this API is Web3 node package. As part of the EVM integration we plan on supporting a limited set of those APIs. The APIs that will not be supported are generally APIs that involve ethereum specific concepts. In order to support the APIs, we plan on creating a JSON RPC server to handle the DApp requests. Using the Go SDK, those requests will then be able to interact with the ledger.



We need to look into how the API can be used to interact with Fabric CA for identity management.
We hope to create a inner grpc layer to be implemented on top of Fabric, and work with Burrow and Sawtooth to share an outer layer.

Create an IPC Provider or Http Provider. Web3 expects to talk to something local to sign and send transactions to the configured Network. This would mean the provider is the only thing that needs to be created to enable web3 support. The provider we create would use an SDK to talk to the peer.

Runtime

This section describes how Fabric interacts with Burrow EVM implementation

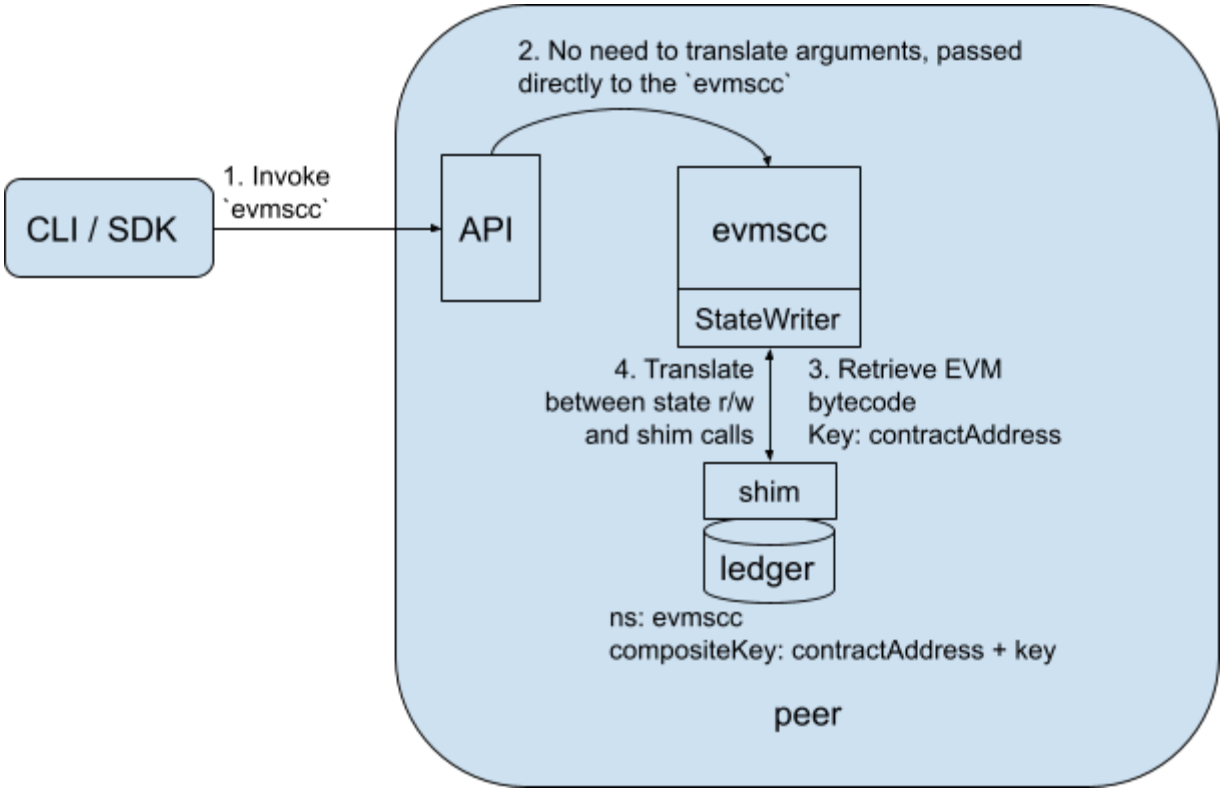
Execution

Burrow EVM library is wrapped in a new system chaincode `evmscc`. ~~User still invokes the chaincode with name:version, e.g. mycc:v1.0. Internally, targeted chaincode is replaced with~~

~~`evmscc`, and original chaincode name is piggybacked to `args`. EVM calls `lscc` to retrieve corresponding contract bytecode with cc name.~~ User invokes the `evmscc` chaincode with the contract address as one of the arguments. We will expect users to understand that they are targeting the `evmscc`.

Bytecode management

The contract bytecode is stored in `lscc` ledger during installation. The contract bytecode will be store in the `evmscc` and will be installed through a transaction, not an installation.



Events

Permissions

Test

This section describes test plans

- E2E test in Fabric
- E2E test using an Ethereum Distributed App

CI Integration