A Deeper Look at Different Smart Contract Platforms

We are living in the era of the smart contract. While Bitcoin may have shown us that a payment system can exist in a decentralized peer-to-peer atmosphere. However, it was with the advent of Ethereum, that the floodgates well and truly opened. Ethereum ushered in the era of the second generation blockchain, and people finally saw the true potential of Dapps and Smart Contracts.

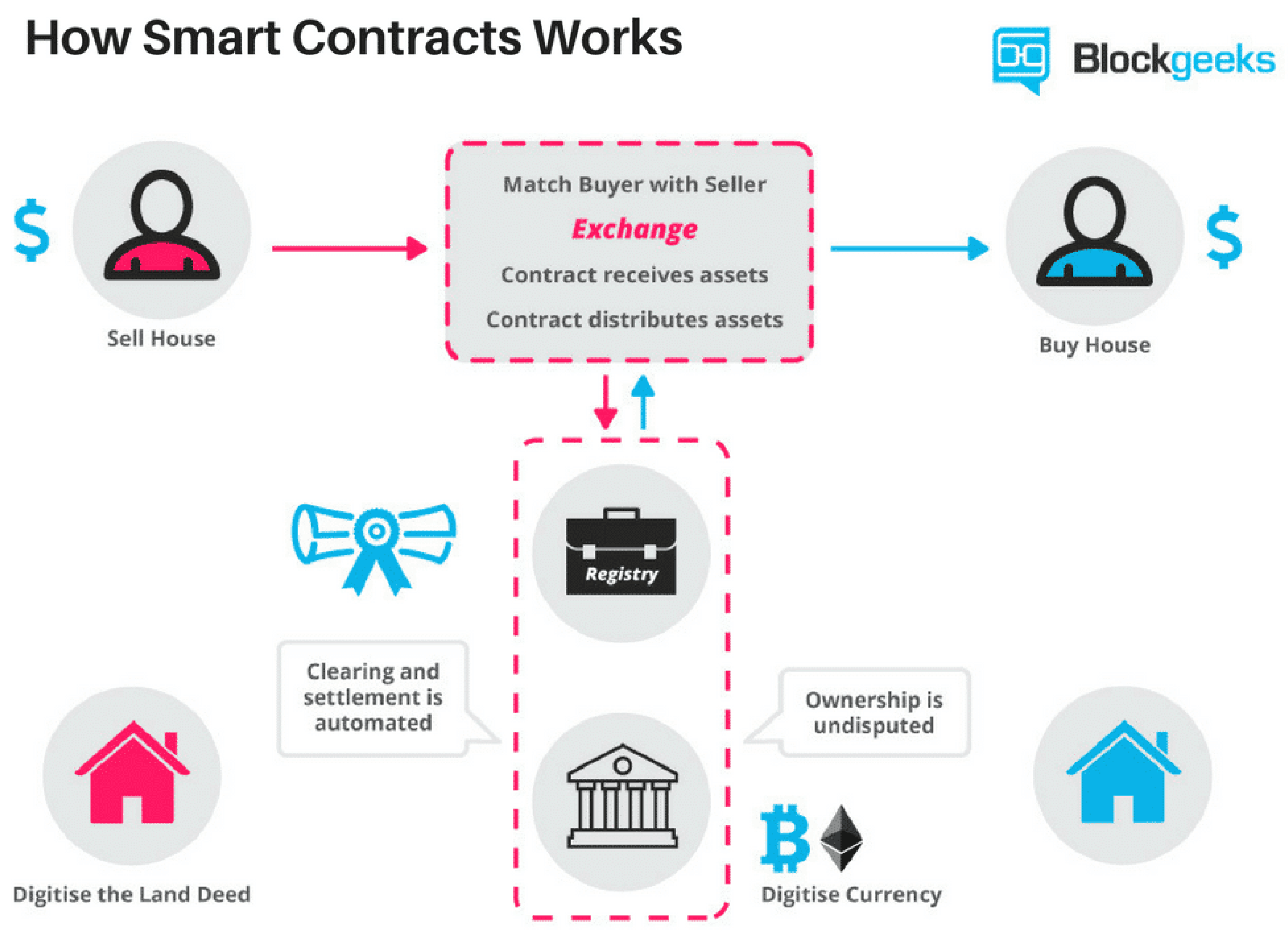
A Deeper Look at Different Smart Contract Platforms

In this guide, we are going to go through some smart contract platforms out there and see what sets them apart. Some of them are already working, while some are under development.

Before we do that though, let's ask ourselves a question.

What exactly are smart contracts?

Smart contracts are automated contracts. They are self-executing with specific instructions written on its code which get executed when certain conditions are made.



You can learn more about smart contracts in our in-depth guide here.

So, what are the desirable properties that we want in our smart contract?

Anything that runs on a blockchain needs to be immutable and must have the ability to run through multiple nodes without compromising its integrity. As a result of which, smart contract functionality needs to be three things:

- **Deterministic.**
- **Terminable.**
- **Isolated.**

Feature #1: Deterministic

A program is deterministic if it gives the same output to a given input every single time. Eg. If 3+1 = 4 then 3+1 will ALWAYS be 4 (assuming the same base). So when a program gives the same output to the same set of inputs in different computers, the program is called deterministic.

There are various moments when a program can act in an un-deterministic manner:

- **Calling un-deterministic system functions**: When a programmer calls an un-deterministic function in their program.

- **Un-deterministic data resources**: If a program acquires data during runtime and that data source is un-deterministic then the program becomes un-deterministic. Eg. Suppose a program that acquires the top 10 google searches of a particular query. The list may keep changing.

- **Dynamic Calls**: When a program calls the second program it is called dynamic calling. Since the call target is determined only during execution, it is un-deterministic in nature.

Feature #2: Terminable

In mathematical logic, we have an error called "halting problem". Basically, it states that there is an inability to know whether or not a given program can execute its function within a time limit. In 1936, Alan Turing deduced, using Cantor's Diagonal Problem, that there is no way to know whether a given program can finish in a time limit or not.

This is obviously a problem with smart contracts because, contracts by definition, must be capable of termination in a given time limit. There are some measures taken to ensure that there is a way to externally "kill" the contract and do not enter into an endless loop which will drain resources:

- **Turing Incompleteness**: A Turing Incomplete blockchain will have limited functionality and not be capable of making jumps and/or loops. Hence they can't enter an endless loop.

- **Step and Fee Meter**: A program can simply keep track of the number "steps" it has taken, i.e. the number of instructions it has executed, and then terminate once a particular step count has been executed.   Another method is the Fee meter. Here the contracts are executed with a pre-paid fee. Every instruction execution requires a particular amount of fee. If the fee spent exceeds the pre-paid fee then the contract is terminated.

- **Timer**: Here a pre-determined timer is kept. If the contract execution exceeds the time-limit then it is externally aborted.

Feature #3: Isolated

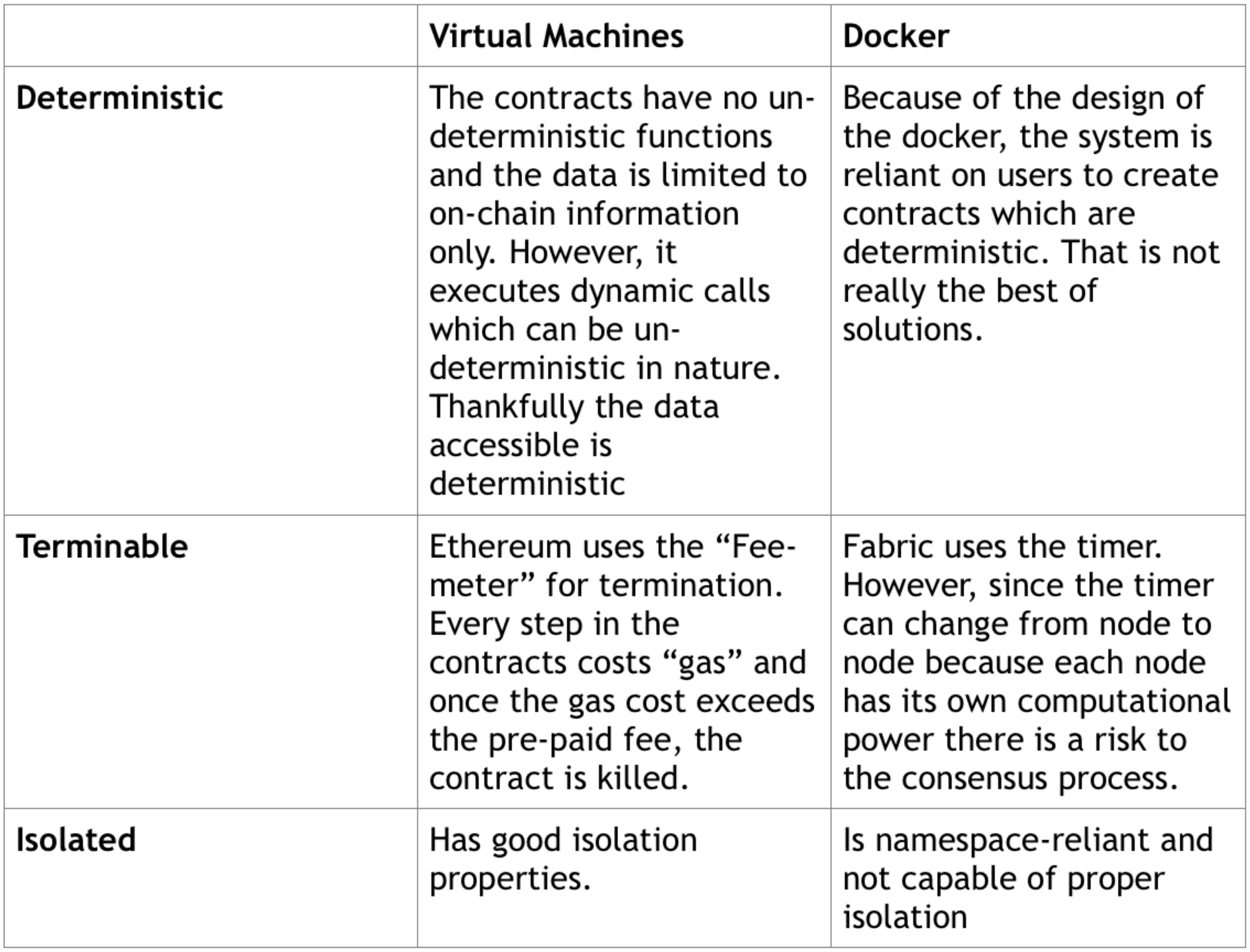
In a blockchain, anyone and everyone can upload a smart contract. However, because of this the contracts may, knowingly and unknowingly contain virus and bugs.

If the contract is not isolated, this may hamper the whole system. Hence, it is critical for a contract to be kept isolated in a sandbox to save the entire ecosystem from any negative effects.

Now that we have seen these features, it is important to know how they are executed. Usually, the smart contracts are run using one of the two systems:

- **Virtual Machines**: Ethereum and Neo use this
- **Docker**: Fabric uses this.

Let's compare these two and determine which makes for a better ecosystem. For simplicity's sake, we are going to compare Ethereum (Virtual Machine) to Fabric (Docker).



	Virtual Machines	**Docker**
Deterministic	The contracts have no un-deterministic functions and the data is limited to on-chain information only. However, it executes dynamic calls which can be un-deterministic in nature. Thankfully the data accessible is deterministic	Because of the design of the docker, the system is reliant on users to create contracts which are deterministic. That is not really the best of solutions.
Terminable	Ethereum uses the "Fee-meter" for termination. Every step in the contracts costs "gas" and once the gas cost exceeds the pre-paid fee, the contract is killed.	Fabric uses the timer. However, since the timer can change from node to node because each node has its own computational power there is a risk to the consensus process.
Isolated	Has good isolation properties.	Is namespace-reliant and not capable of proper isolation

So, as can be seen, Virtual Machines provide better Deterministic, terminable and isolated environment for the Smart contracts.

Ok, so now we know what smart contracts are and the fact that Virtual machines are better platforms for smart contracts. Let's look at what exactly do Dapps require to run efficiently.

What do Dapps require?

Or, to frame it more specifically, what does a DAPP require to be successful and a hit with the mainstream audience? What are its absolute minimum requirements?

Support for Millions of Users

It should be scalable enough for millions of users to use it. This is especially true for DAPPs that are looking for mainstream acceptance.

Free Usage

The platform should enable the devs to create Dapps which are free to use for their users. No user should have to pay the platform to gain the benefits of a Dapp.

Easily Upgradable

The platform should allow the developers the freedom to upgrade the Dapp as and when they want. Also, if some bug does affect the Dapp, the devs should be able to fix the DAPP without affecting the platform.

Low Latency

A DAPP should run as smoothly as possible and with the lowest possible latency.

Parallel Performance

A platform should allow their Dapps to be processed parallelly in order to distribute the workload and save up time.

Sequential Performance

However, not all the functions on a blockchain should be done that way. Think of transaction execution itself. Multiple transactions can’t be executed in parallel; it needs to be done one at a time to avoid errors like double spends.

So, what are the platforms available to us when it comes to DAPP creation?

BitShares and Graphene have good throughput but are definitely not smart contract suitable.
Ethereum is clearly the most obvious choice in the market. It has amazing smart contract abilities but the low transaction speed is a major issue. Plus, the gas price can be problematic as well.

Ok, so now that we know what Dapps require, let’s go through some smart contract platforms.

We will be looking at:

- Ethereum
- EOS
- Stellar

- Cardano
- Neo
- Hyperledger Fabric

Ethereum

First and foremost, we have Ethereum, the one that started it all.

This is how Ethereum's website defines it:

"Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third party **interference. These apps run on a custom built blockchain, an enormously powerful shared global infrastructure that can move value around and represent the ownership of property."**

But in simpler terms, Ethereum is planning to be the ultimate software platform of the future. If the future is decentralized and Dapps become commonplace, then Ethereum has to be the front and center of it.

The Ethereum Virtual Machine or EVM is the virtual machine in which all the smart contracts function in Ethereum. It is a simple yet powerful Turing Complete 256-bit virtual machine. Turing Complete means that given the resources and memory, any program executed in the EVM can solve any problem.

In order to code smart contracts in the EVM, one needs to learn the programming language Solidity.

Solidity is a purposefully slimmed down, loosely-typed language with a syntax very similar to ECMAScript (Javascript). There are some key points to remember from the Ethereum Design Rationale document, namely that we are working within a stack-and-memory model with a 32-byte instruction word size, the EVM (Ethereum Virtual Machine) gives us access to the program "stack" which is like a register space where we can also stick memory addresses to make the Program Counter loop/jump (for sequential program control), an expandable temporary "memory" and a more permanent "storage" which is actually written into the permanent blockchain, and most importantly, the EVM requires total determinism within the smart contracts.

So, before we continue, let's check out a basic Solidity contract example. (Codes were taken from github).

Let's run a simple while loop in solidity:

```
contract BasicIterator

{

address creator; // reserve one "address"-type spot
```

```
uint8[10] integers; // reserve a chunk of storage for 10 8-bit unsigned integers in an
array

function BasicIterator()

{

creator = msg.sender;

uint8 x = 0;

//Section 1: Assigning values

while(x < integers.length) {

integers[x] = x;

x++;

} }

function getSum() constant returns (uint) {

uint8 sum = 0;

uint8 x = 0;

//Section 2: Adding the integers in an array.

while(x < integers.length) {
sum = sum + integers[x];

x++;
}

return sum;

}

// Section 3: Killing the contract

function kill()

{

if (msg.sender == creator)

{

suicide(creator);
}

}

}
```

So, let's analyze the code. For ease of understanding, we have divided the code into 3 sections.

Section 1: Assigning Values

In the first step, we are filling up an array called "integers" which takes in 10 8-bit unsigned integers. The way we are doing it is via a while loop. Let's look at what is happening inside the while loop.

```
while(x < integers.length) {

integers[x] = x;

x++;

}
```

Remember, we have already assigned a value of "0" to the integer x. The while loop goes from 0 to integers.length. Integers.length is a function which returns the max capacity of the array. So, if we decided that an array will have 10 integers, arrayname.length will return a value of 10. In the loop above, the value of x goes from 0 – 9 (<10) and assigns the value of itself to the integers array as well. So, at the end of the loop, integers will have the following value:

0,1,2,3,4,5,6,7,8,9.

Section 2: Adding the array content

Inside the getSum() function we are going to add up the contents of the array itself. The way are going to do it is by repeating the same while loop as above and using the variable "sum" to add the contents of the array.

Section 3: Killing the contract

This function kills the contract and sends the remaining funds in the contract back to the contract creator.

What is Gas?

"Gas" is the lifeblood of the Ethereum ecosystem, there is no other way of putting that. Gas is a unit that measures the amount of computational effort that it will take to execute certain operations.

Every single operation that takes part in Ethereum, be it a simple transaction, or a smart contract, or even an ICO takes some amount of gas. Gas is what is used to calculate the number of fees that need to be paid to the network in order to execute an operation.

When someone submits a smart contract, it has a pre-determined gas value. When the contract is executed each and every step of the contract requires a certain amount of gas to execute.

This can lead to two scenarios:

1. *The gas required is more than the limit set. If that's the case then the state of the contract is reverted back to its original state and all the gas is used up.*

2.The gas required is less than the limit set. If that's the case, then the contract is completed and the leftover gas is given over to the contract setter.

While Ethereum may have paved the way for smart contracts. It does face some scalability issues. However, innovations such as plasma, raiden, sharding etc. may solve this issue.

EOS

EOS are aiming to become a decentralized operating system which can support industrial-scale decentralized applications.

That sounds pretty amazing but what has really captured the public's imagination is the following two claims:

- They are planning to completely remove transaction fees.

Related Guides

What Are Dapps? The New Decentralized Future

What is Tokenomics? Ultimate Investor's Guide -Part 1

Understand Blockchain Business Models: Complete Guide

- They are claiming to have the ability to conduct millions of transactions per second.

These two features are the reasons why Dapp developers are fascinated with EOS. Let's look at how EOS achieves both of these things.

Removal of Fees

EOS works on an ownership model whereby users own and are entitled to use resources proportional to their stake, rather than having to pay for every transaction. So, in essence, if you hold N tokens of EOS then you are entitled to N*k transactions. This, in essence, eliminates transaction fees.

The costs of running and hosting applications on Ethereum can be high for a developer who wants to test their application on the blockchain. The gas price involved in the early stages of development can be enough to turn off new developers.

The fundamental difference between the way Ethereum and EOS operate is that while Ethereum rents out their computational power to the developers, EOS gives ownership of their resources. So, in essence, if you own

1/1000th of the stake in EOS then you will have ownership of 1/1000th of the total computational power and resources in EOS.

As ico-reviews states in their article:

“EOS’s ownership model provides DAPP developers with predictable hosting costs, requiring them only to maintain a certain percentage or level of stake, and makes it possible to create freemium applications. Furthermore, since EOS token holders will be able to rent/delegate their share of resources to other developers, the ownership model ties the value of EOS tokens to the supply and demand of bandwidth and storage.”

Increased Scalability

EOS gets its scalability from its DPOS consensus mechanism. DPOS stands for delegated proof of stake and this is how it works:

Firstly, anyone who holds tokens on a blockchain integrated into the EOS software can select the block producers through a continuous approval voting system. Anyone can participate in the block producer election and they will be given an opportunity to produce blocks proportional to the total votes they receive relative to all other producers.

How does it work?

- Blocks are produced in the rounds of 21.

- At the start of every round 21 block producers are chosen. Top 20 are automatically chosen while the 21st one is chosen proportional to the number of their votes relative to the other producers.

- The producers are then shuffled around using a pseudorandom number derived from the block time. This is done to ensure that a balance of connectivity to all other producers is maintained.

- To ensure that regular block production is maintained and that block time is kept to 3 seconds, producers are punished for not participating by being removed from consideration. A producer has to produce at least one block every 24 hours to be in consideration.

Since there are so few people involved in the consensus, it is faster and more centralized than Ethereum and Bitcoin, which uses the entire network for consensus.

The WASM Language

EOS uses WebAssembly aka WASM programming language. The reason why they use it is because of its following properties (taken from webassembly.org):

- **Speed and Efficiency**: WebAssembly executes at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.

- **Open and Debuggable**: It is designed to be pretty-printed in a textual format for debugging, testing, experimenting, optimizing, learning, teaching, and writing programs by hand.

- **Safe**: WebAssembly describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines.

EOS is the perfect platform to create industrial scale Dapps. Let’s imagine that you are creating a decentralized Twitter. If you created that on Ethereum, then the user would have to spend some gas whilst executing each and every step of a tweet.

If you did the same thing in EOS, users won’t need to spend gas because transaction fees are 0! However, since EOS is not as decentralized as Ethereum, Dapps that require high-degrees of censorship resistance may not be a good fit for it.

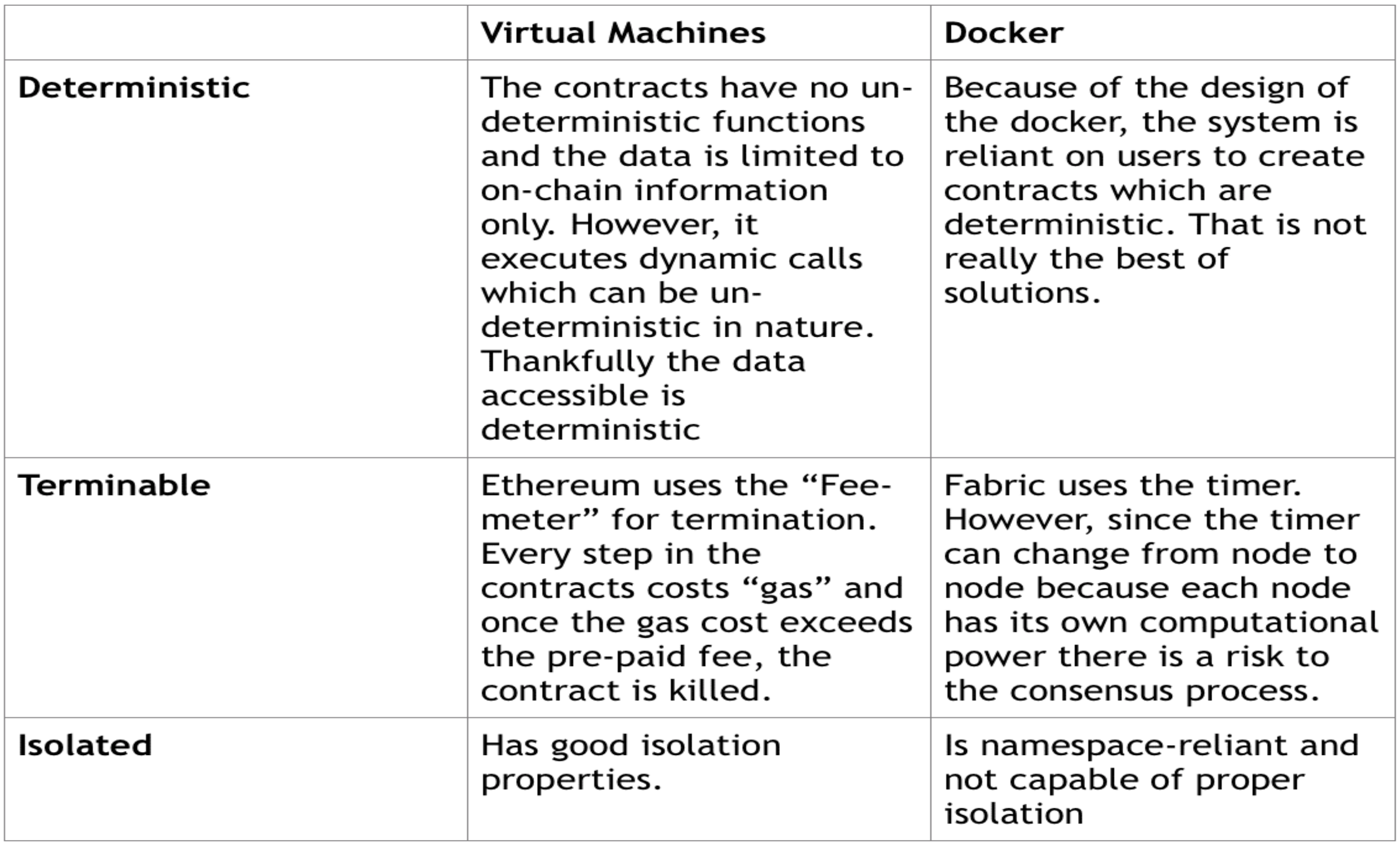
Stellar

Stellar is the brainchild of Jed McCaleb and Joyce Kim was formed back in 2014 when it was forked from the Ripple protocol. Stellar, according to their website,

“is a platform that connects banks, payments systems, and people. Integrate to move money quickly, reliably, and at almost no cost”.

Using Stellar, one can move money across borders quickly, reliably, and for fractions of a penny.

Unlike Ethereum, Stellar Smart Contracts (SSC) are not Turing complete. The following table gives you a good idea of the differences between Stellar and Ethereum smart contracts:



	Virtual Machines	**Docker**
Deterministic	The contracts have no un-deterministic functions and the data is limited to on-chain information only. However, it executes dynamic calls which can be un-deterministic in nature. Thankfully the data accessible is deterministic	Because of the design of the docker, the system is reliant on users to create contracts which are deterministic. That is not really the best of solutions.
Terminable	Ethereum uses the “Fee-meter” for termination. Every step in the contracts costs “gas” and once the gas cost exceeds the pre-paid fee, the contract is killed.	Fabric uses the timer. However, since the timer can change from node to node because each node has its own computational power there is a risk to the consensus process.
Isolated	Has good isolation properties.	Is namespace-reliant and not capable of proper isolation

Some stats may pop up straight away.

Most notably, the 5 second confirmation time and the fact that a single transaction on the Stellar network costs only ~$0.0000002!

```
$stellarNetwork->buildTransaction($customerKeypair)
->addCreateAccountOp($escrowKeypair, 100.00006) // 100 XLM after setup fees + transfer
transaction
->submit($customerKeypair);
}

print "Created escrow account: " . $escrowKeypair->getPublicKey() . PHP_EOL;

/*
* In order to make this an escrow account, we need to prove to the worker that
* no one is able to withdraw funds from it while the worker is searching for
* a vanity address.
*
* This is accomplished by:
* - Making the worker and customer signers of equal weight (1)
* - Requiring both signers to agree on any transaction (thresholds are set to 2)
*
* However, we also need to handle the case where no worker takes the job and we
* need to reclaim the account. This can be done by adding a preauthorized merge
* transaction that's not valid until 30 days from now.
*
* This allows the worker to know that the funds are guaranteed to be available
* for 30 days.
*/

// Load up the escrow account
$account = $stellarNetwork->getAccount($escrowKeypair);

// Precalculate some sequence numbers since they're necessary for transactions
$startingSequenceNumber = $account->getSequence();
// Track how many transactions are necessary to set up the escrow account
// We need this so we can correctly calculate the "reclaim account" sequence number
$numSetupTransactions = 5;

$reclaimAccountOrPaySeqNum = $startingSequenceNumber + $numSetupTransactions + 1;

// Update the account with a data value indicating what vanity address to search for
print "Adding data entry to request a vanity address...";
$stellarNetwork->buildTransaction($escrowKeypair)
->setAccountData('request:generateVanityAddress', 'G*ZULU')
->submit($escrowKeypair);
print "DONE" . PHP_EOL;

// Fallback transaction: reclaim the escrow account if no workers generate the
// vanity address in 30 days
$reclaimTx = $stellarNetwork->buildTransaction($escrowKeypair)
```

```
->setSequenceNumber(new BigInteger($reclaimAccountOrPaySeqNum))
// todo: uncomment this out in a real implementation
//->setLowerTimebound(new \DateTime('+30 days'))
->setAccountData('request:generateVanityAddress')
->addMergeOperation($customerKeypair)
->getTransactionEnvelope();

// Add hash of $reclaimTx as a signer on the account
// See: https://www.stellar.org/developers/guides/concepts/multi-sig.html#pre-authorized-
transaction
$txHashSigner = new Signer(
SignerKey::fromPreauthorizedHash($reclaimTx->getHash()),
2 // weight must be enough so no other signers are needed
);
$addReclaimTxSignerOp = new SetOptionsOp();
$addReclaimTxSignerOp->updateSigner($txHashSigner);

print "Adding pre-authorized reclaim transaction as a signer... ";
$stellarNetwork->buildTransaction($escrowKeypair)
->addOperation($addReclaimTxSignerOp)
->submit($escrowKeypair);
print "DONE" . PHP_EOL;

print "Added pre-auth reclaim transaction valid at sequence " .
$reclaimAccountOrPaySeqNum . PHP_EOL;
print "To reclaim the escrow account, run 90-reclaim-escrow.php" . PHP_EOL;

// Add worker account as a signer of weight 1
$workerSigner = new Signer(
SignerKey::fromKeypair($workerKeypair),
1 // requires another signer
);
$addSignerOp = new SetOptionsOp();
$addSignerOp->updateSigner($workerSigner);
$stellarNetwork->buildTransaction($escrowKeypair)
->addOperation($addSignerOp)
->submit($escrowKeypair);

// Add customer account as second signer of weight 1
$workerSigner = new Signer(
SignerKey::fromKeypair($customerKeypair),
1 // requires another signer
);
$addSignerOp = new SetOptionsOp();
$addSignerOp->updateSigner($workerSigner);
$stellarNetwork->buildTransaction($escrowKeypair)
->addOperation($addSignerOp)
->submit($escrowKeypair);

// Increase thresholds and set master weight to 0
// All operations now require threshold of 2
```

```
$thresholdsOp = new SetOptionsOp();
$thresholdsOp->setLowThreshold(2);
$thresholdsOp->setMediumThreshold(2);
$thresholdsOp->setHighThreshold(2);
$thresholdsOp->setMasterWeight(0);
$stellarNetwork->buildTransaction($escrowKeypair)
->addOperation($thresholdsOp)
->submit($escrowKeypair);

print PHP_EOL;
print "Finished configuring escrow account" . PHP_EOL;
```

Cardano

One of the most interesting projects to have come out is Cardano. Similar to Ethereum, Cardano is a smart contract platform however, Cardano offers scalability and security through layered architecture. Cardano's approach is unique in the space itself since it is built on scientific philosophy and peer-reviewed academic research

Cardano aims to increase scalability via their Ouroboros proof of stake consensus mechanism. In order to code smart contracts in Cardano, you will need to use Plutus, which is based on Haskell, the language used to code Cardano.

While C++ and most traditional languages are Imperative programming languages, Plutus and Haskell are functional programming languages.

So, how does functional programming work?

Suppose there is a function f(x) that we want to use to calculate a function g(x) and then we want to use that to work with a function h(x). Instead of solving all of those in a sequence, we can simply club all of them together in a single function like this:

h(g(f(x)))

This makes the functional approach easier to reason mathematically. This is why functional programs are supposed to be a more secure approach to smart contract creation. This also aids in simpler Formal Verification which pretty much means that it is easier to mathematically prove what a program does and how it acts out. This gives Cardano its "High Assurance Code" property.

Let’s take a real-life example of this and see why it can become extremely critical and even life-saving in certain conditions.

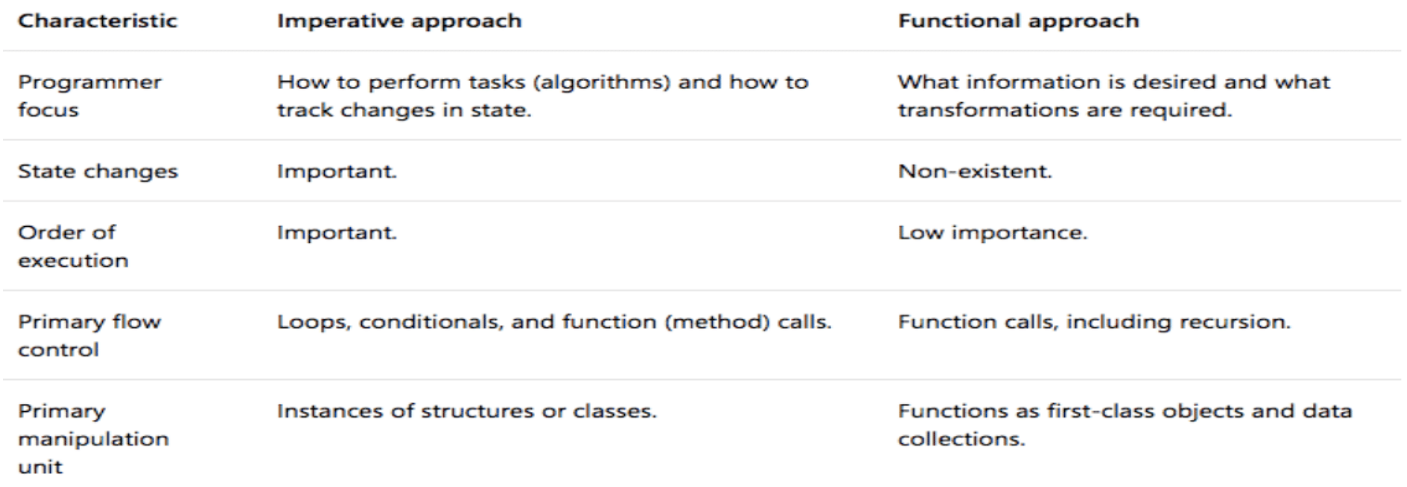
Suppose, we are coding a program that controls air-traffic.

As you can imagine, coding such a system requires a high degree of precision and accuracy. We can’t just blindly code something and hope for the best when people’s lives are at risk. In situations like these, we need a code that can be proven to work to a high degree of mathematical certainty.

This is precisely why the functional approach is so desirable.

And that is exactly what Cardano is using Haskell to code their ecosystem and Plutus for their smart contracts. Both Haskell and Plutus are functional languages.

The following table compares the Imperative approach with the Functional approach.



Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

So, let’s let’s look at the advantages of the functional approach:

- Helps with creating high assurance code because it is easier to mathematically prove how the code is going to behave.

- Increases the readability and maintainability because each function is designed to accomplish a specific task. The functions are also state-independent.

- The code is easier to refractor and any changes in the code are simpler to implement. This makes reiterative development easier.

- The individual functions can be easily isolated which makes them easier to test out and debug.

Neo

Neo, formerly known as Antshares, is often known as the “Ethereum of China”.

According to their website, Neo is a “non-profit community-based blockchain project that utilizes blockchain technology and digital identity to digitize assets, to automate the management of digital assets using smart contracts, and to realize a “smart economy” with a distributed network.”

Neo’s main aim is to be the distributed network for “smart economy”. As their website states:

Digital Assets + Digital Identity + Smart Contract = Smart Economy.

Neo was developed by Shanghai based blockchain R&D company “OnChain”. Onchain was founded by CEO Da Hongfei and CTO Erik Zhang. Research on Neo started around 2014. In 2016, Onchain was listed in the Top 50 Fintech Company in China by KPMG.

Neo wanted to create a smart contract platform which has all the advantages of an Ethereum Virtual Machine, without crippling their developers with language barriers. In ethereum, you will need to learn solidity to code smart contracts, while in Neo, you can even use Javascript to code smart contracts.

Neo Smart Contract 2.0

Neo’s smart contract system aka The Smart Contract 2.0 has three parts to it:

- NeoVM.
- InteropService
- DevPack

NeoVm

This is a pictorial representation of the Neo Virtual Machine:

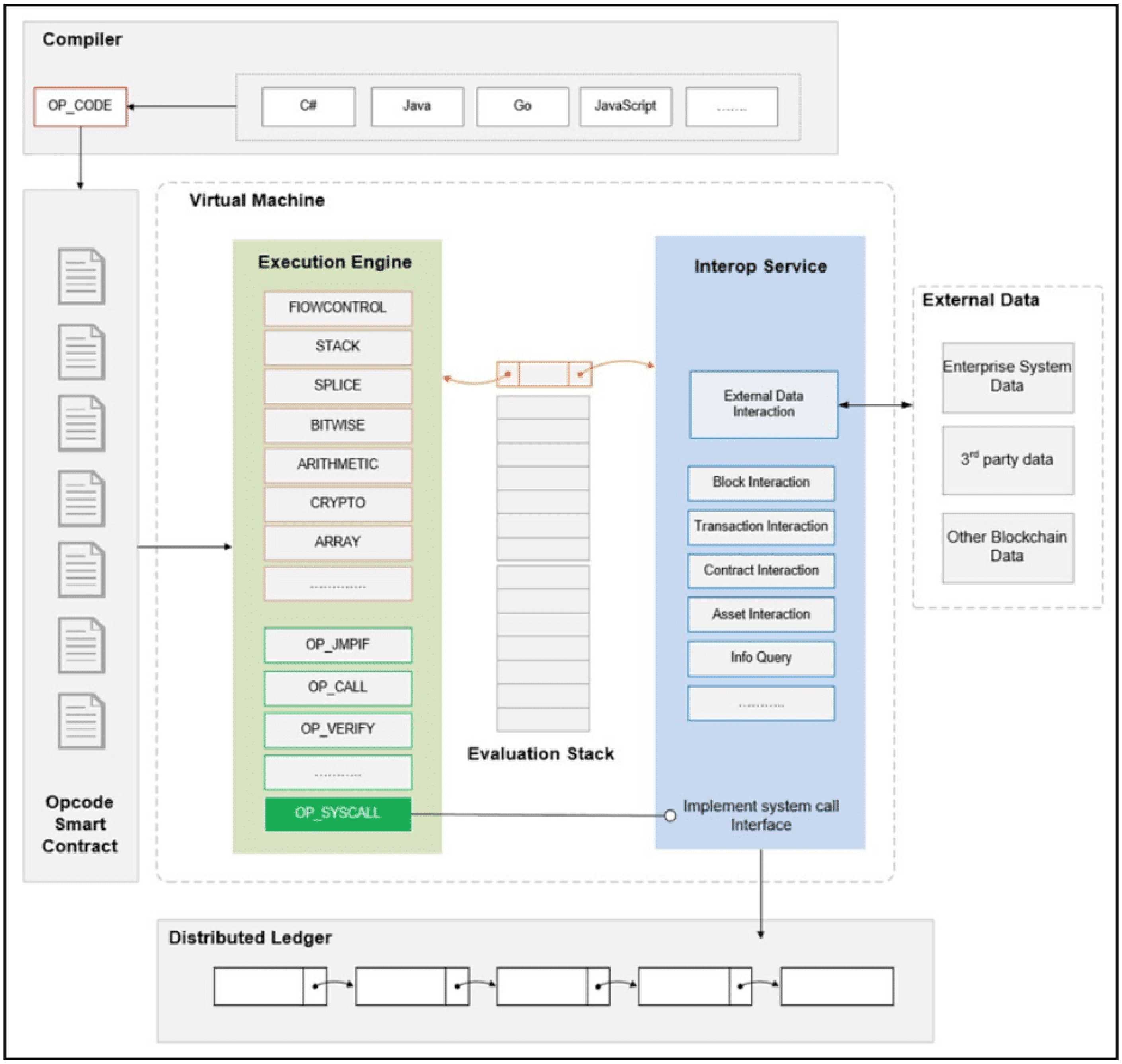


Image Credit: Neo Whitepaper

As the Neo Whitepaper states, the NeoVM or Neo Virtual Machine is a lightweight, general-purpose VM whose architecture closely resembles JVM and .NET Runtime. It is similar to a virtual CPU that reads and executes instructions in the contract in sequence, performs process control based on the functionality of the instruction operations, logic operations and so on. It is versatile with a good start-up speed which makes it a great environment to run smart contracts.

InteropService

The InteropService increases the utility of the smart contracts. It allows the contracts to access data outside the NeoVM without compromising on the overall stability and efficiency of the system.

Currently, the interoperable service layer provides some APIs for accessing the chain-chain data of the smart contract. The data that it can access are:

- Block information.
- Transaction information
- Contract information.
- Asset information

….among others.

It also provides storage space for smart contracts.

DevPack

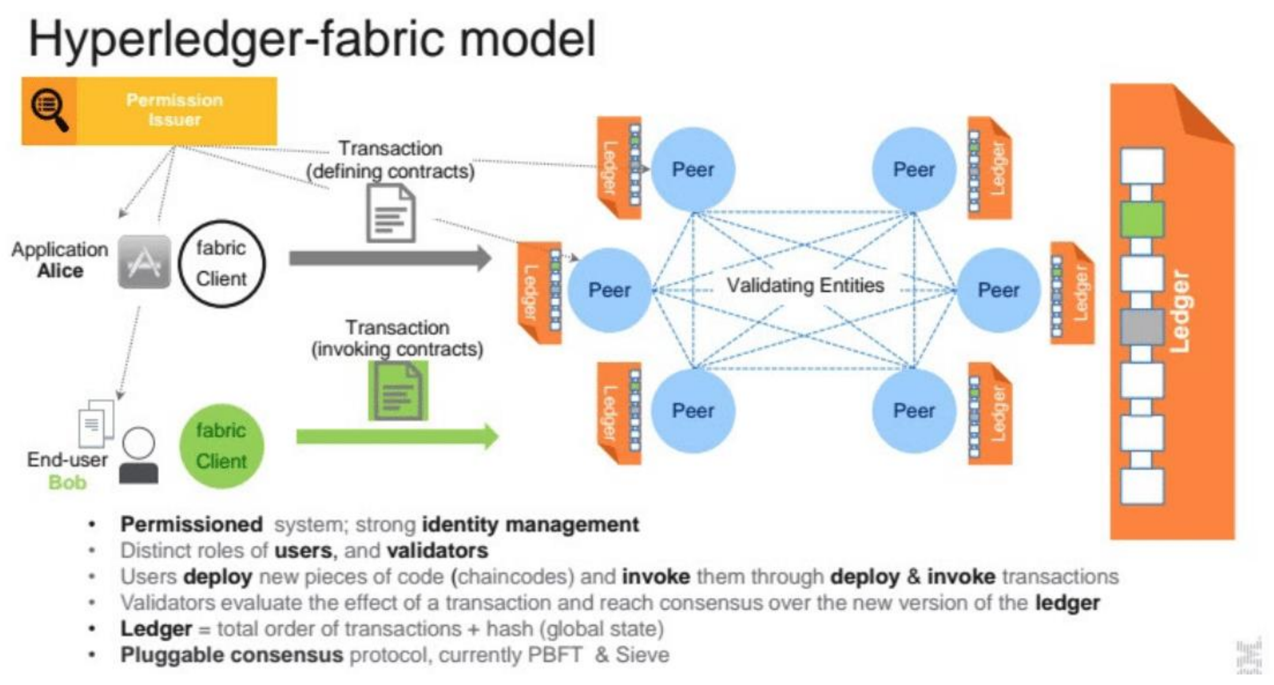
DevPack includes the high-level language compiler and the IDE plug-in. Since the NeoVM architecture is pretty similar to JVM and .NET Runtime, it enables contracts to be coded in other languages. As you can imagine, this greatly reduced the time taken by developers to learn how to create smart contracts.

Hyperledger Fabric

According to their website, “Hyperledger is an open source collaborative effort created to advance cross-industry blockchain technologies. It is a global collaboration, hosted by The Linux Foundation, including leaders in finance, banking, Internet of Things, supply chains, manufacturing, and Technology.”

Maybe the most interesting project in the Hyperledger family is IBM’s Fabric. Rather than a single blockchain Fabric is a base for the development of blockchain based solutions with a modular architecture.

With Fabric different components of Blockchains, like consensus and membership services can become plug-and-play. Fabric is designed to provide a framework with which enterprises can put together their own, individual blockchain network that can quickly scale to more than 1,000 transactions per second.



Hyperledger-fabric model

- **Permissioned** system; strong **identity management**
- Distinct roles of **users**, and **validators**
- Users **deploy** new pieces of code (chaincodes) and **invoke** them through **deploy & invoke** transactions
- Validators evaluate the effect of a transaction and reach consensus over the new version of the **ledger**
- **Ledger** = total order of transactions + hash (global state)
- **Pluggable consensus** protocol, currently PBFT & Sieve

What is Fabric and how does it work? The framework is implemented in Go. It is made for enabling consortium blockchains with different degrees of permissions. Fabric heavily relies on a smart contract system called Chaincode, which every peer of the networks runs in Docker containers.

In order to write Chaincode, one must be well-versed in four functions:

- PutState: Create new asset or update existing one.
- GetState: Retrieve asset.
- GetHistoryForKey : Retrieve history of changes.
- DelState: ‘Delete’ asset.

Following is an example of a Chaincode:

```
// Define the Smart Contract structure
type SmartContract struct { }

// Define the car structure, with 4 properties.
type Car struct {
Make string `json:"make"`
Model string `json:"model"`
Colour string `json:"colour"`
Owner string `json:"owner"`
}

/*
* The Invoke method is called as a result of an application request to run the Smart
Contract "fabcar"
```

```
* The calling application program has also specified the particular smart contract
function to be called, with arguments
*/
func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {

// Retrieve the requested Smart Contract function and arguments
function, args := APIstub.GetFunctionAndParameters()

// Route to the appropriate handler function to interact with the ledger appropriately
if function == "initLedger" {
return s.initLedger(APIstub)
} else if function == "createCar" {
return s.createCar(APIstub, args)
}

return shim.Error("Invalid Smart Contract function name.")
}

func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.Response {
return shim.Success([]byte("Ledger is now running, success!"))
}

// Add new car to database with obtained arguments
func (s *SmartContract) createCar(APIstub shim.ChaincodeStubInterface, args []string)
sc.Response {

if len(args) != 5 {
return shim.Error("Incorrect number of arguments. Expecting 5")
}

var car = Car{Make: args[1], Model: args[2], Colour: args[3], Owner: args[4]}
carAsBytes, _ := json.Marshal(car)

APIstub.PutState(args[0], carAsBytes)

return shim.Success(nil)
}

func main() {
// Create a new Smart Contract
err := shim.Start(new(SmartContract))

if err != nil {
fmt.Printf("Error creating new Smart Contract: %s", err)
}
}
```

Conclusion

So, there you have it. Some of the smart contract platforms and the various properties which make them unique. There is no “one-size-fits-all”, at least for now. You will need to choose the platform that best suits the functionalities required for your Dapp.