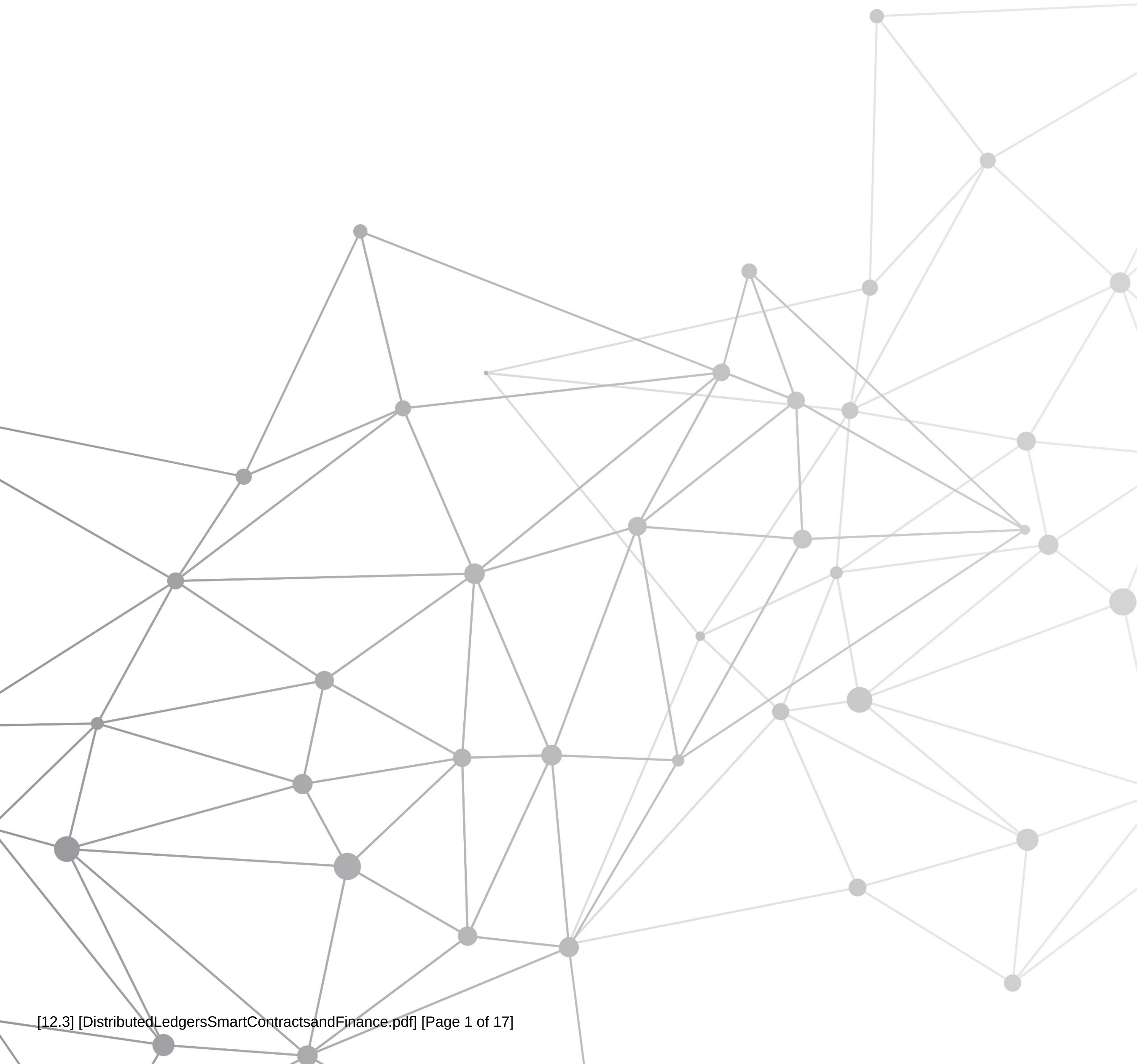
symbiont

Smart Securities,™ Intelligent Markets

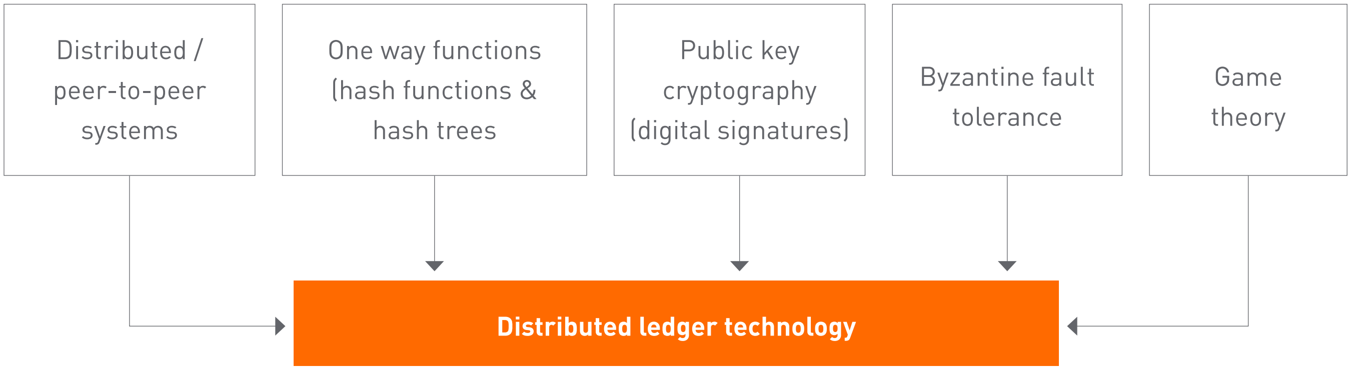


Distributed ledgers, smart contracts, and finance

Symbiont (www.symbiont.io) is a technology infrastructure company developing distributed ledger and smart contract-based products that address the needs of institutional finance. In this document, we aim to provide an overview of these emerging technologies and discuss how they can be applied within a financial context. Our primary goal is to assist the technical or semi-technical reader to better understand the innovative and disruptive value present, and following that, to inspire further discussion and research.

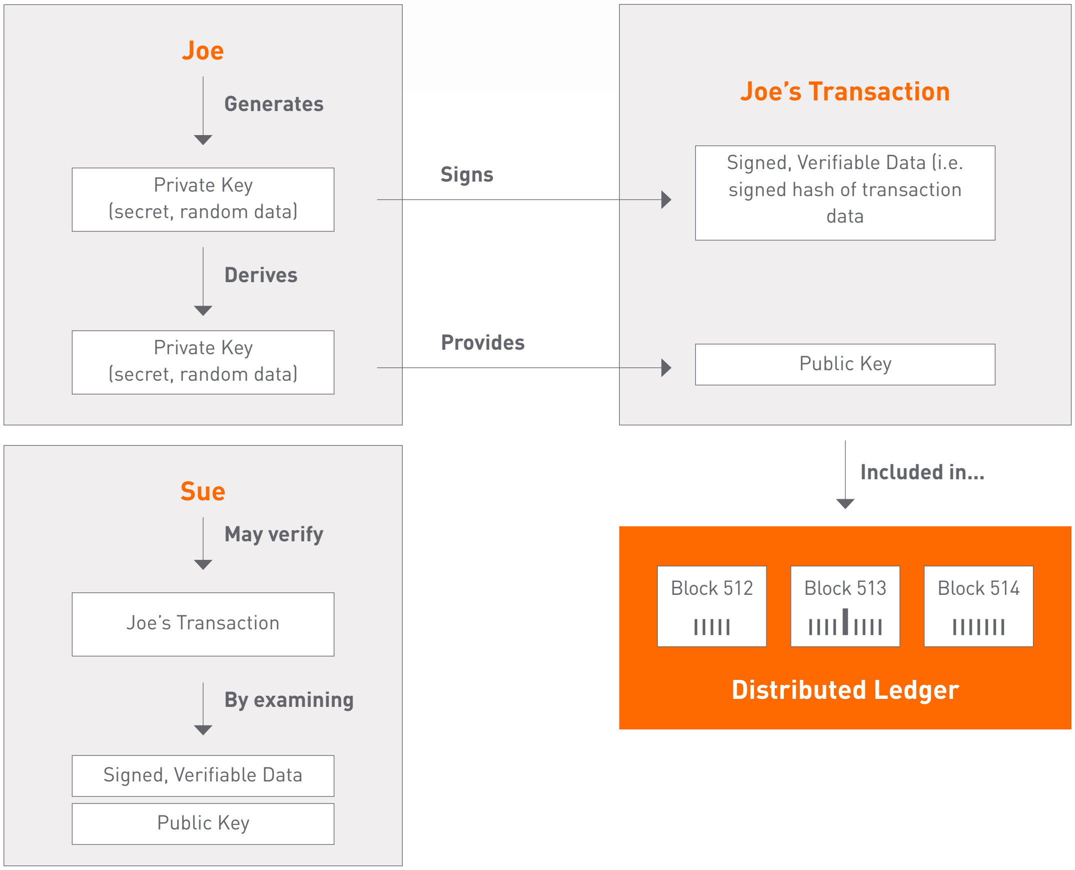
Distributed ledger and consensus systems

Distributed ledger technology is a synthesis of multiple pre-existing developments in computer science:


Distributed / peer-to-peer systems
One way functions (hash functions & hash trees
Public key cryptography (digital signatures)
Byzantine fault tolerance
Game theory
Distributed ledger technology

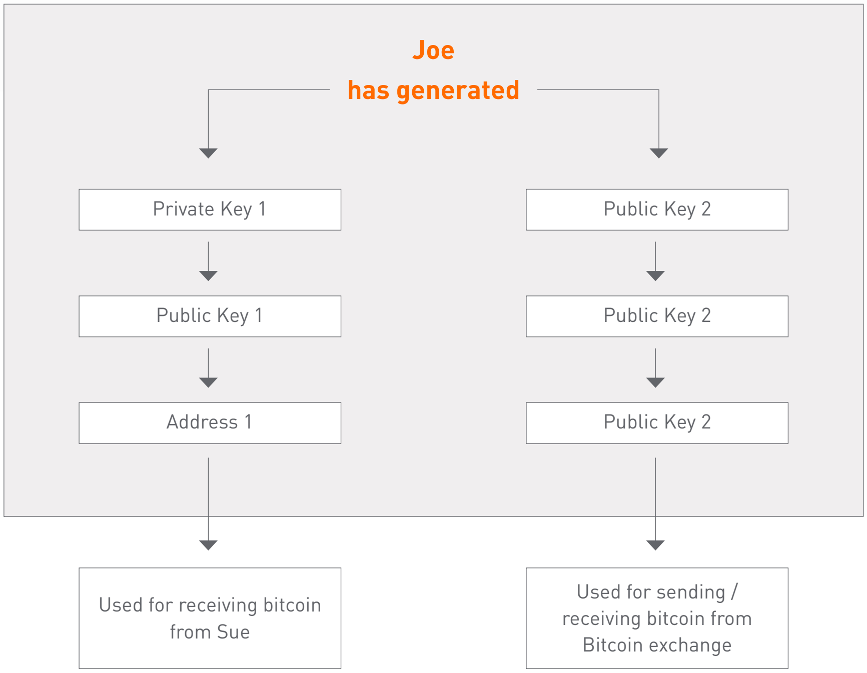
A distributed ledger can be thought of as a strictly ordered, shared database that only allows data to be added to it. Independent copies of the database are hosted by multiple parties, and a **peer-to-peer** protocol is employed to allow each party to add to the database as well as to validate the other parties' additions. Distributed ledgers are appropriate in cases where the data to be stored has a natural (e.g. time-based) ordering and the participating parties are either untrusted or semi-trusted by each other. Through the mechanics of the ledger, the parties seek a state known as **consensus**, which reflects a state of agreement by the parties regarding the data and its ordering. The peer-to-peer protocol employed to accomplish this is known as a **consensus protocol**.

Each party may make a new entry into the database that modifies its state somehow (i.e. a **transaction**) by cryptographically signing the data they want to add with their **private key**, and then **broadcasting** the resultant data to the peer to peer **network**, along with their **public key**.



Once the transaction is broadcast, other parties on the network may then easily verify the authenticity of the broadcasting party, as well as the integrity of the data itself. This is because the party's public key is known and is derived from the (secret) private key the broadcasting party has in their possession, and it is computationally impossible to construct a private key from a public key.

Similar to the Internet's concept of an IP **address**, the broadcasting party normally has its operations associated with a specific address, which is derived from their public and private keys, and commonly appears as a long string of letters and numbers (e.g. "17eLG11sGPJ3bfY43GxGSRu8bGbVR5mRY7"). These addresses are **pseudonymous** (i.e. semi-anonymous) because they weakly identify the party. That is, if you were to just see an address, you wouldn't know which party owned it. However, this doesn't mean the party will always remain anonymous, as the address could be associated to them through their own disclosure, inferred via its relationship to an address that is known, and so on. A given party can create and own multiple addresses, and use them for different purposes.

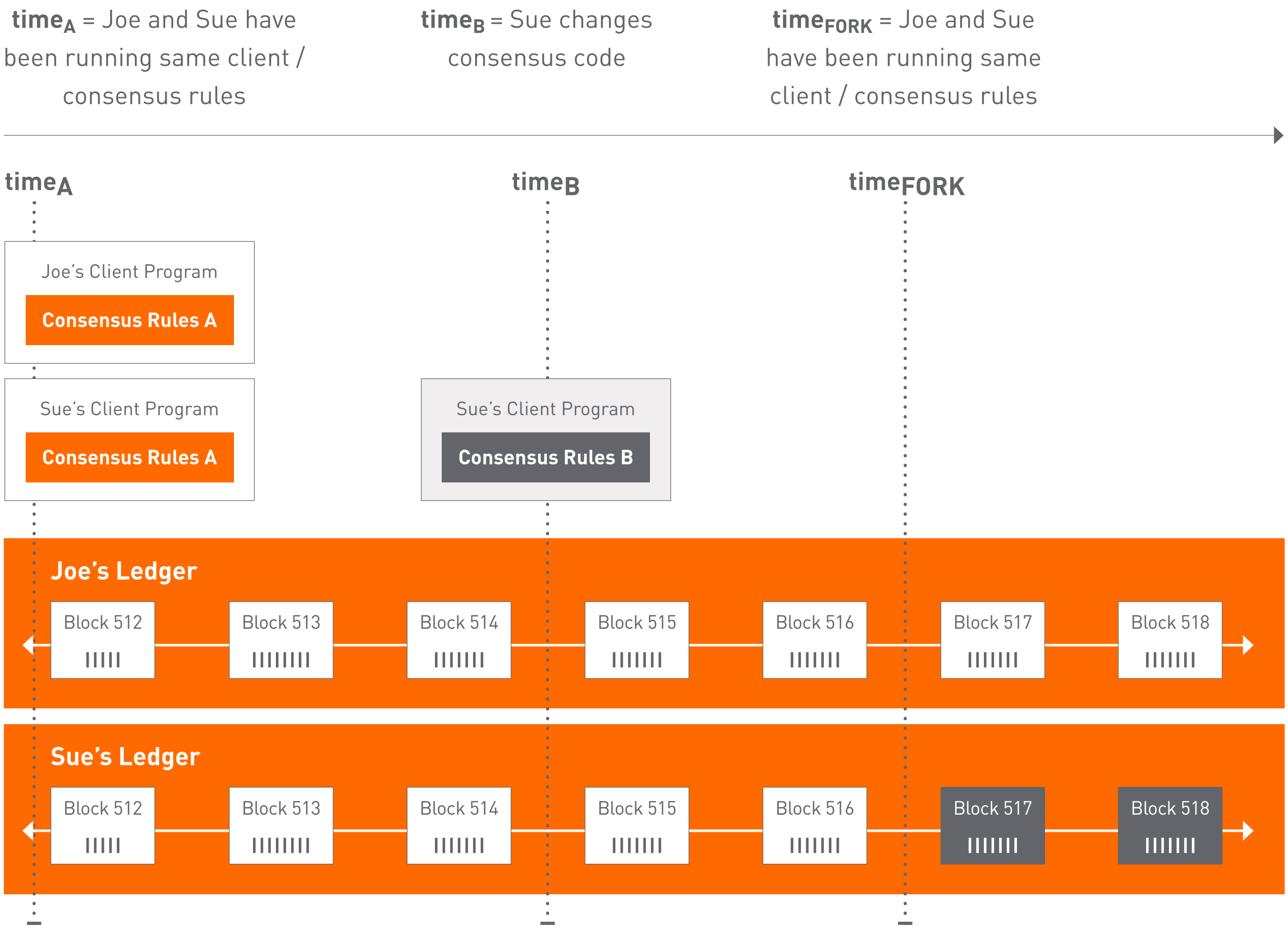

Joe
has generated
Private Key 1
Public Key 1
Address 1
Public Key 2
Public Key 2
Public Key 2
Used for receiving bitcoin from Sue
Used for sending / receiving bitcoin from Bitcoin exchange

Beyond the application of cryptographic signing to verify the identity and permit actions of network participants, cryptographic hashes (i.e. one-way functions) are employed to ensure consistency of the ledger as a whole. In the case of Bitcoin and similar distributed ledger designs, the term blockchain is used because the ledger is actually a series of groupings (i.e. blocks) of transactions. Each block contains one or more transactions, and each block is linked, or "chained" as it were, to the previous block by containing a special hash of that previous block's data (which itself will hash its parent block, and so on, all the way back to the very first block in the ledger). This allows any network participant to easily verify their copy's transaction ordering within each block, and the ordering of blocks as a whole.

Distributed ledger vs "blockchain": It's important to note that not all distributed ledgers are blockchains. The existence of a block derives from Bitcoin's so-called Nakamoto consensus protocol. Other kinds of distributed ledgers, such as those based on alternative consensus algorithms like Practical Byzantine Fault Tolerance (PBFT) -- which pre-dated Nakamoto consensus -- do not utilize the concept of blocks, and simply work with an ordered list of transactions.

Each network participant, or node, starts up and runs a client program to join and function in the peer-to-peer distributed ledger network. The consensus rules (i.e. the software logic that implements the consensus protocol) are a component of the client program, and comprise a significant portion of it, with the remaining being supporting code that is not strictly part of the consensus protocol. Being able to establish and maintain consensus, where all parties copies of the ledger are consistent, requires that all nodes have compatible versions of the consensus rules. Maintaining consensus also means that all consensus rules must be deterministic. This means that given the same input, the rules will always produce the same output. This is because each party (or an algorithmically determined representative subset of parties) will normally verify and run through every transaction on the network, in parallel and in duplicate. With matching (or at least compatible), deterministically executing consensus rules, network transactions can be published, validated and relayed consistently across nodes.

In practice, network nodes may be on different versions of the client program, which may have differences in the consensus rules and/or non-consensus-sensitive supporting code. With differences in the supporting code, this is not an issue. However, if a network node operates a version of the consensus rules that differs from some or all of their peers in a non-backwards or forwards-compatible fashion, or utilizes non-deterministic code, it could lead to a **network fork**. This is where the node's copy of the ledger will diverge from that or their peers, into a parallel version of the data.


time_A = Joe and Sue have been running same client / consensus rules

time_B = Sue changes consensus code

time_{FORK} = Joe and Sue have been running same client / consensus rules

With several distributed ledger designs (e.g. "open" ledgers like Bitcoin), minor forks happen rather commonly as a result of normal operation. These are operational in nature and not due to consensus protocol differences. Moreover, they are automatically resolved by the protocol over a short period of time. A network fork due to consensus rule mismatch is a different thing: in this case, the operators of the affected node(s) must resolve this problem by updating their client to an appropriate compatible version of the software. If the nodes can't agree on whose ledger is the "right" ledger and maintain their differing consensus rules, then the fork becomes permanent, and the two parallel copies of the ledger persist and will differ increasingly over time.

Public Networks

With **open networks** like Bitcoin, where essentially anyone may participate anonymously, consensus rules are generally more complex and oftentimes utilize the expense of large amounts of computational power from a subset of the network's participants. This so called **proof-of-work** mechanism acts as the basis of a system that serves to secure the network from malicious actors who may try things like forging, delaying or rolling back transactions.

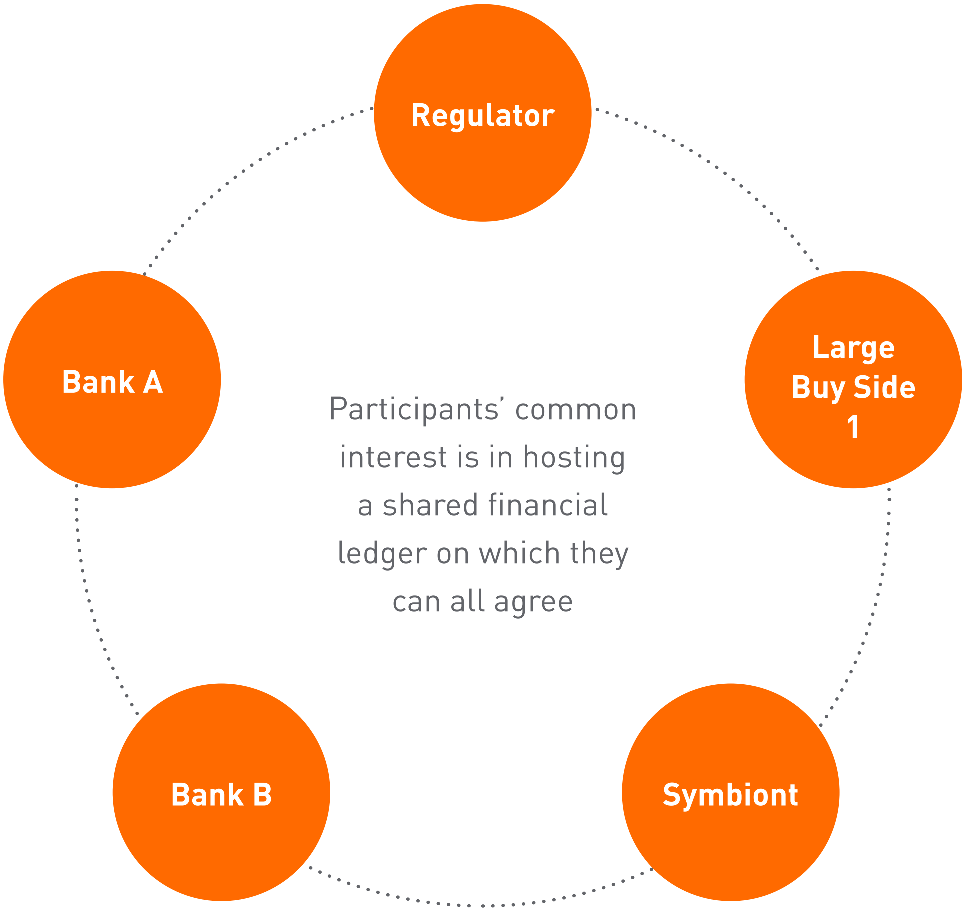
With proof-of-work, this subset of network participants (called **miners**) commits their hardware and electricity to securing the network, and in doing so earns a chance to append a new block to the ledger. Their resources are committed to generating repeated hashes, such that the hashed data value generated is under some other value (the **difficulty**). In order to ensure a consistent generation rate of new blocks, this difficulty value is modified by the network consensus rules so that a solution to the problem is found at roughly equal intervals, on average. By imposing a resource cost to participate, proof-of-work effectively protects the network against **Sybil attacks**, which are where a malicious party could cheaply create numerous fake identities and utilize them to manipulate outcomes on the peer-to-peer network.

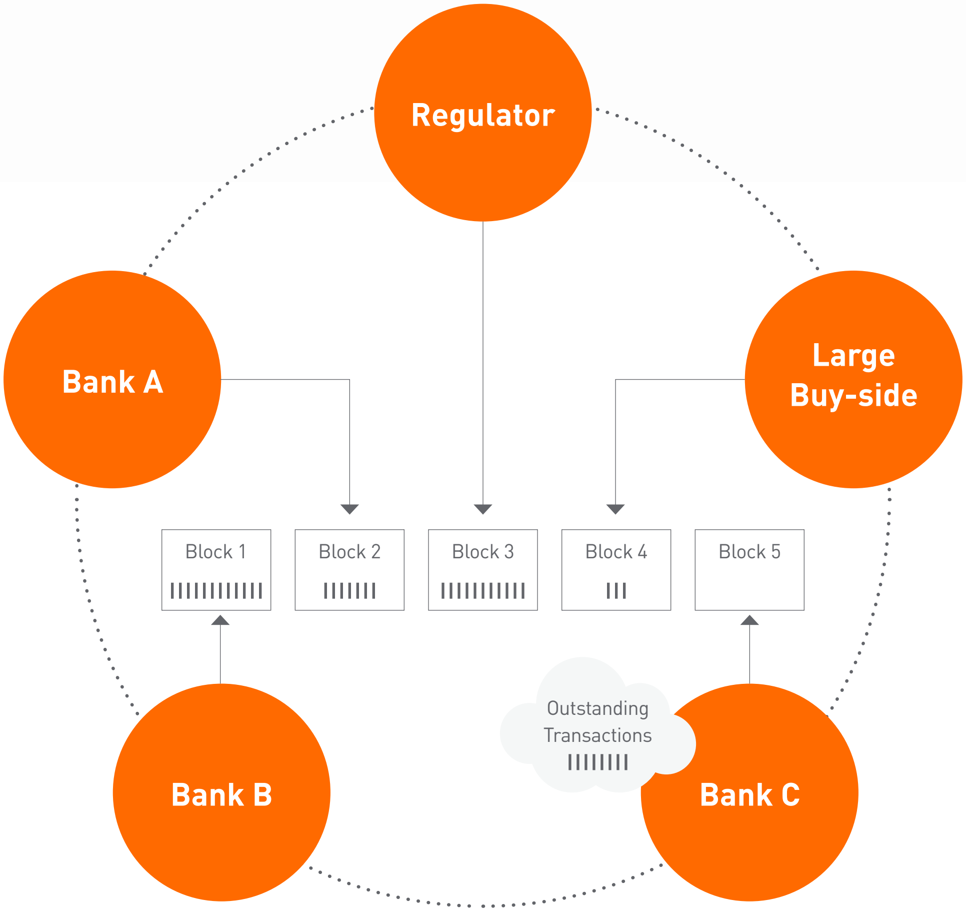
Along with containing some or all of the outstanding (**uncommitted**) transactions, newly mined block will also contain a special transaction that awards the suitable miner with some quantity of newly created **value token** (e.g. bitcoin). Thus, the value token itself acts as an incentivisation mechanism to encourage behavior that both protects and operates the network. As long as the majority of miners are not acting against their economic best interests (i.e. to compete for value token), the network remains secure, and new blocks are properly generated.

Permissioned networks

In contrast with open networks, **permissioned networks** require that their participants be known to each other and verified in some form. With permissioned networks, no anonymous participants may participate. An example of an appropriate use-case would be a banking syndicate, whose members may maintain a common ledger tracking transactions for a particular asset class.

Proof-of-work consensus algorithms in use with open networks provide a mechanism for eliminating the risk of an effective Sybil attack. With an open network, Sybil attacks are a concern because anonymous participants may create numerous nodes and act maliciously. In a permissioned network (e.g. with a list of whitelisted nodes, which every node in the network has a copy of), proof-of-work is not strictly necessary, as there is no real possibility of a Sybil attack.


Regulator
Large Buy Side 1
Bank A
Participants' common interest is in hosting a shared financial ledger on which they can all agree
Bank B
Symbiont



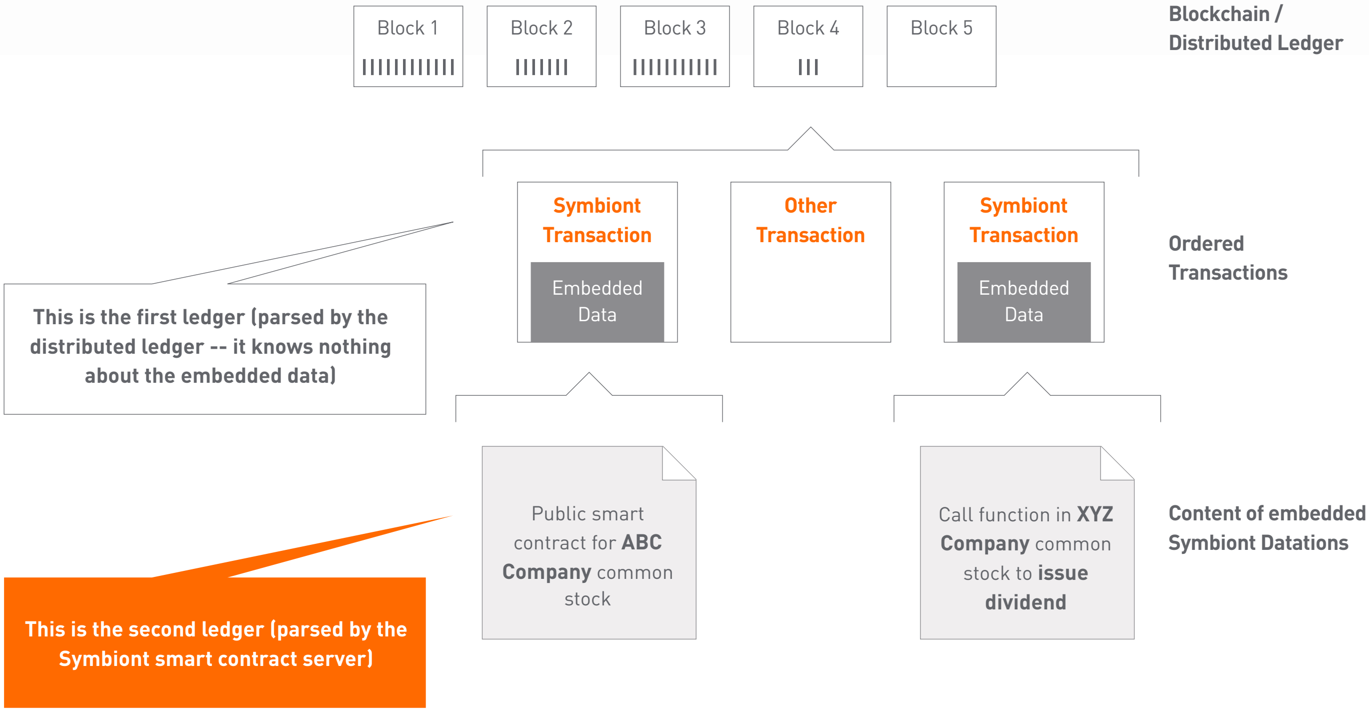
White-listed nodes, appending to the ledger in some deterministic order.

In place of proof-of-work, more simplified consensus algorithms (such as PBFT, mentioned earlier) can be employed in a permissioned distributed ledger design to ensure that any of the participants can easily verify the integrity of the ledger and trace any dishonest, malicious or errant activity to a particular, pre-identified participant. A combination of both technical and legal measures could then be utilized to resolve the issue.

Due to the more controllable, homogenous hardware and bandwidth provisioning of a permissioned network, along with the simpler consensus protocol designs which can be utilized, higher transaction processing throughput can be more easily obtained.

Embedded consensus

The team behind Symbiont are pioneers of a technology known as **embedded consensus**. An embedded consensus system records data into a distributed ledger in a special way. To the distributed ledger consensus protocol itself, this data is not used and appears to be arbitrary. However, the embedded consensus system employs an additional protocol level that sits alongside the distributed ledger. For each new transaction, the embedded consensus system will detect and parse out this data payload, and then interpret it according to its own rules. Thus, this approach essentially constitutes a ledger-within-a-ledger, where one ledger is embedded within the other and parsed independently:


Block 1
Block 2
Block 3
Block 4
Block 5
Blockchain / Distributed Ledger
Symbiont Transaction
Embedded Data
Other Transaction
Symbiont Transaction
Embedded Data
Ordered Transactions
This is the first ledger (parsed by the distributed ledger -- it knows nothing about the embedded data)
Public smart contract for ABC Company common stock
Call function in XYZ Company common stock to issue dividend
Content of embedded Symbiont Datations
This is the second ledger (parsed by the Symbiont smart contract server)

Symbiont incorporates this technique in the construction of its Smart Securities™ system, which operates out-of-the-box as an embedded ledger within its own distributed ledger implementation. In practice, this technique allows for certain benefits:

Modularity: Embedded consensus enforces the desirable concepts of data abstraction, modularity and encapsulation. This reduces potential implementation and operational risks over a more tightly commingled implementation. Our approach is conceptually similar to how the TCP/IP protocol suite is organized into multiple layers, with the higher level layers' data being embedded into the respective lower level ones as a packet is created and sent "on the wire".

Data security: Data embedded into the distributed ledger inherits the same level of security as any other transaction in the distributed ledger. This is because transactions with this embedded data appear like any other transaction to the underlying distributed ledger.

Flexibility: Modular **translation layer** logic may be written to allow the same smart contract code to be used with virtually any distributed ledger. This brings massive flexibility benefits and reduces the risk of distributed ledger technology lock-in. Symbiont's Smart Securities™ system incorporates such a translation layer and can run on virtually any distributed ledger system.

Simplicity: Symbiont's embedded consensus technology is dramatically simpler than other competing approaches in virtually all cases. Instead of a more monolithic approach, it achieves this by focusing on permissioned execution of smart contract logic, leaving the actual transaction generation, packaging and consensus operations to that of the distributed ledger itself.

Well developed & tested: Counterparty, an embedded consensus platform for Bitcoin that was developed by the Symbiont founding team members, was one of the first projects to apply this approach. With Counterparty, embedded consensus allowed it to quickly overcome inherent limitations in the core Bitcoin protocol so that new functionality could be "layered on". Since its creation, Counterparty has been in production use for over two years on the Bitcoin network, powering on average between one-half and one percent of daily Bitcoin transactions. The Counterparty reference implementation has been subjected to numerous security and code audits by well-known distributed ledger researchers, and it's well-tested code formed the basis of the embedded consensus implementation in the Symbiont technology stack.

Benefits of distributed ledger and consensus systems to finance

Distributed ledgers -- for the first time ever -- allow property to be fully represented digitally. This is a huge innovation, as before this there was no fool-proof way to model something like a house deed (which can only have one owner at a time, and is transferred from one party to another) in a fully digital fashion. Thus, the common rails of the Internet, which have had such an impact on things like news media, music, commerce, and more, can now handle bonafide property as well: deeds, titles, stock, bonds, cash and equivalents, loans, et cetera.

Applied to finance, distributed ledger technology naturally can eliminate the vast swaths of redundancy, manual process and systems complexity present today. For instance, consider the approach for most market deployments, where each party maintains its own separate database systems (essentially, ledgers) with interoperability protocols like FIX, SWIFT and FpML used to communicate changes and exchange data between them. With certain asset types, such as syndicated loans, each party may have a dozen or more separate ledgers. This then becomes a huge operational headwind, as massive amounts of personnel and manual process is required to ensure validity and consistency of the data between participants.

Applying distributed ledger technology to this problem we are able to, over time, reduce the number of ledgers in use to essentially just one, across all parties combined. Each party, then, has permissioned, role-based access to publish and interact with that single ledger, and the duplication of data is eliminated. The parties gain the efficiency benefits of centralization, without any of the trust requirement downsides. Simply by virtue of publishing data to a distributed ledger, the following additional benefits materialize:

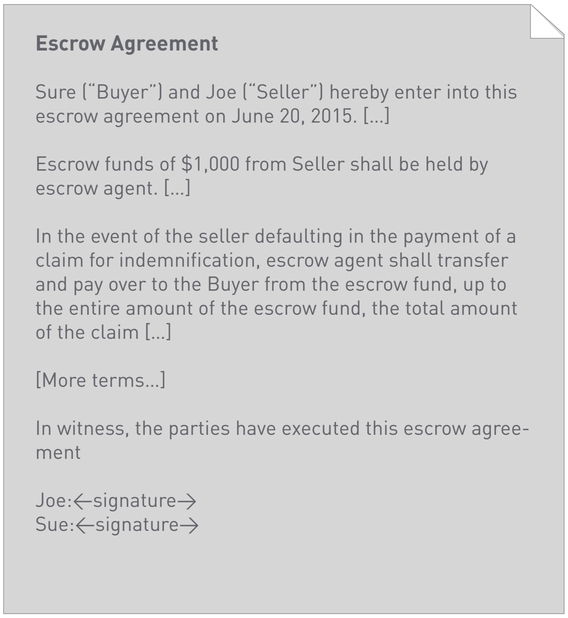
- back-office costs and the burden of internal controls required are massively reduced,
- settlement times are drastically reduced,
- data security is enhanced,
- real-time forecasting and visibility is possible, and
- certain oftentimes desirable practices, such as gross settlement, become possible.

Smart contract technology, layered onto the distributed ledger, drives further benefits, which we'll explore next.

Smart contract technology

The term **smart contract** describes a computer-based protocol that can be used to both define some sort of legal contract, as well as enforce its performance. So, instead of having a contract like normally we do today -- in some written language that's intended for human consumption -- we can have a computer program, in which the terms of a contract can be specified programmatically, as well as the means for enforcing those terms.

Legal Contract



Escrow Agreement

Sure ("Buyer") and Joe ("Seller") hereby enter into this escrow agreement on June 20, 2015. [...]

Escrow funds of $1,000 from Seller shall be held by escrow agent. [...]

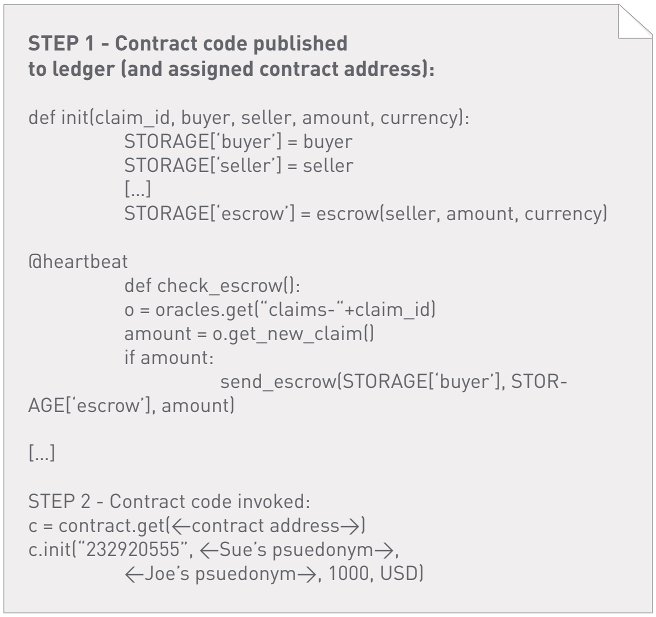
In the event of the seller defaulting in the payment of a claim for indemnification, escrow agent shall transfer and pay over to the Buyer from the escrow fund, up to the entire amount of the escrow fund, the total amount of the claim [...]

[More terms...]

In witness, the parties have executed this escrow agreement

Joe:←signature→
Sue:←signature→

Smart Contract



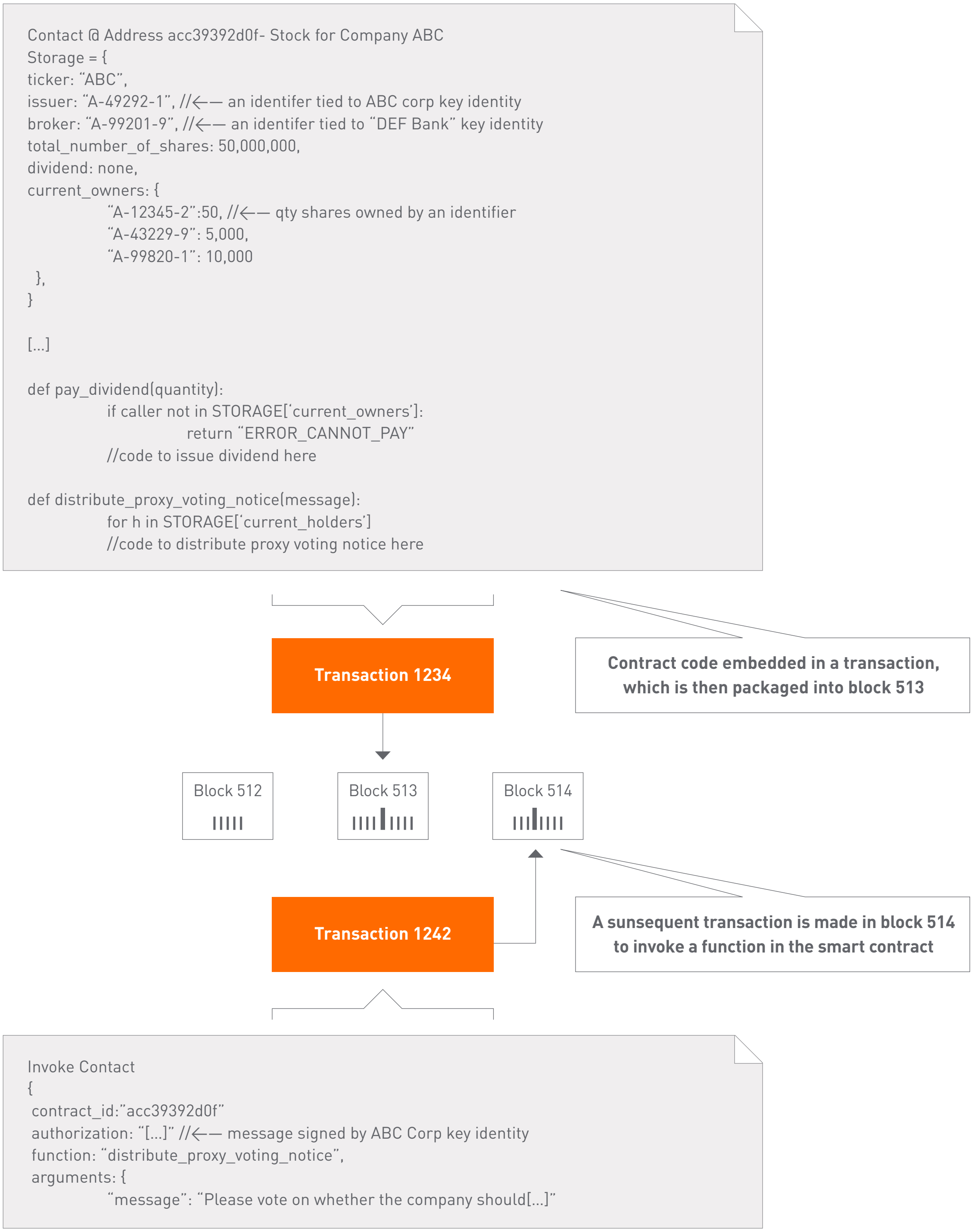
STEP 1 - Contract code published to ledger (and assigned contract address):

```
def init(claim_id, buyer, seller, amount, currency):
            STORAGE['buyer'] = buyer
            STORAGE['seller'] = seller
            [...]
            STORAGE['escrow'] = escrow(seller, amount, currency)

@heartbeat
            def check_escrow():
            o = oracles.get("claims-"+claim_id)
            amount = o.get_new_claim()
            if amount:
                        send_escrow(STORAGE['buyer'], STOR-
AGE['escrow'], amount)

[...]

STEP 2 - Contract code invoked:
c = contract.get(←contract address→)
c.init("232920555", ←Sue's psuedonym→,
            ←Joe's psuedonym→, 1000, USD)
```

As smart contracts can basically be thought of as computer-language legal contracts, it's important that they be stored and operate in a way that resists tampering, provides them with a way to enforce their terms, and permits all parties to a contract to verify those terms and their subsequent carrying out by the contract itself. A distributed ledger becomes the perfect medium for smart contracts to be utilized: it is quite robust, and can provide the means by which a smart contract can control or hold value and thus enforce its terms. Value, then, can be represented either through the use of value tokens, or via proxy tokens, which are tokens representing value stored elsewhere. Although they normally require the injection of a counterparty to provide or manage the underlying external asset, **proxy tokens** are especially useful when dealing with assets that are not and/or cannot "live and die" on the ledger, such as tons of corn, or currency that's presently administered outside of a distributed ledger (US dollars, for instance).

It's important to note that Bitcoin itself actually includes its own simple contract language, loosely based on the functional programming language called Forth. This language (simply called "script") allows for basic smart contracts such as **multisig**, which facilitates transactions where some combination of one or more private keys may be required to spend funds. Multisig operations normally take the form of **M-of-N**, where M signers out of N potential signers are required to authorize a transaction. An example would be a 3-of-5 multisig, where three of five total signers are required to first sign a transaction before they can spend the funds encoded in it.
Bitcoin Script itself is rather restrictive, and is not **Turing complete** in that it notably lacks support for looping. Recently, turing-complete smart contracts have been pioneered by a few groups (including Symbiont and Ethereum) which allow for arbitrary functionality to be represented in a user-friendly general purpose programming language.

Here’s a diagram of a simplified Symbiont turing-complete smart contract, first being embedded into one block, and then being invoked:



```
Contact @ Address acc39392d0f- Stock for Company ABC
Storage = {
ticker: “ABC”,
issuer: “A-49292-1”, //<–– an identifer tied to ABC corp key identity
broker: “A-99201-9”, //<–– an identifer tied to “DEF Bank” key identity
total_number_of_shares: 50,000,000,
dividend: none,
current_owners: {
        “A-12345-2”:50, //<–– qty shares owned by an identifier
        “A-43229-9”: 5,000,
        “A-99820-1”: 10,000
 },
}

[...]

def pay_dividend(quantity):
        if caller not in STORAGE[‘current_owners’]:
                return “ERROR_CANNOT_PAY”
        //code to issue dividend here

def distribute_proxy_voting_notice(message):
        for h in STORAGE[‘current_holders’]
        //code to distribute proxy voting notice here
```

Contract code embedded in a transaction, which is then packaged into block 513

A sunsequent transaction is made in block 514 to invoke a function in the smart contract

```
Invoke Contact
{
 contract_id:”acc39392d0f”
 authorization: “[...]” //<–– message signed by ABC Corp key identity
 function: “distribute_proxy_voting_notice”,
 arguments: {
        “message”: “Please vote on whether the company should[...]”
```

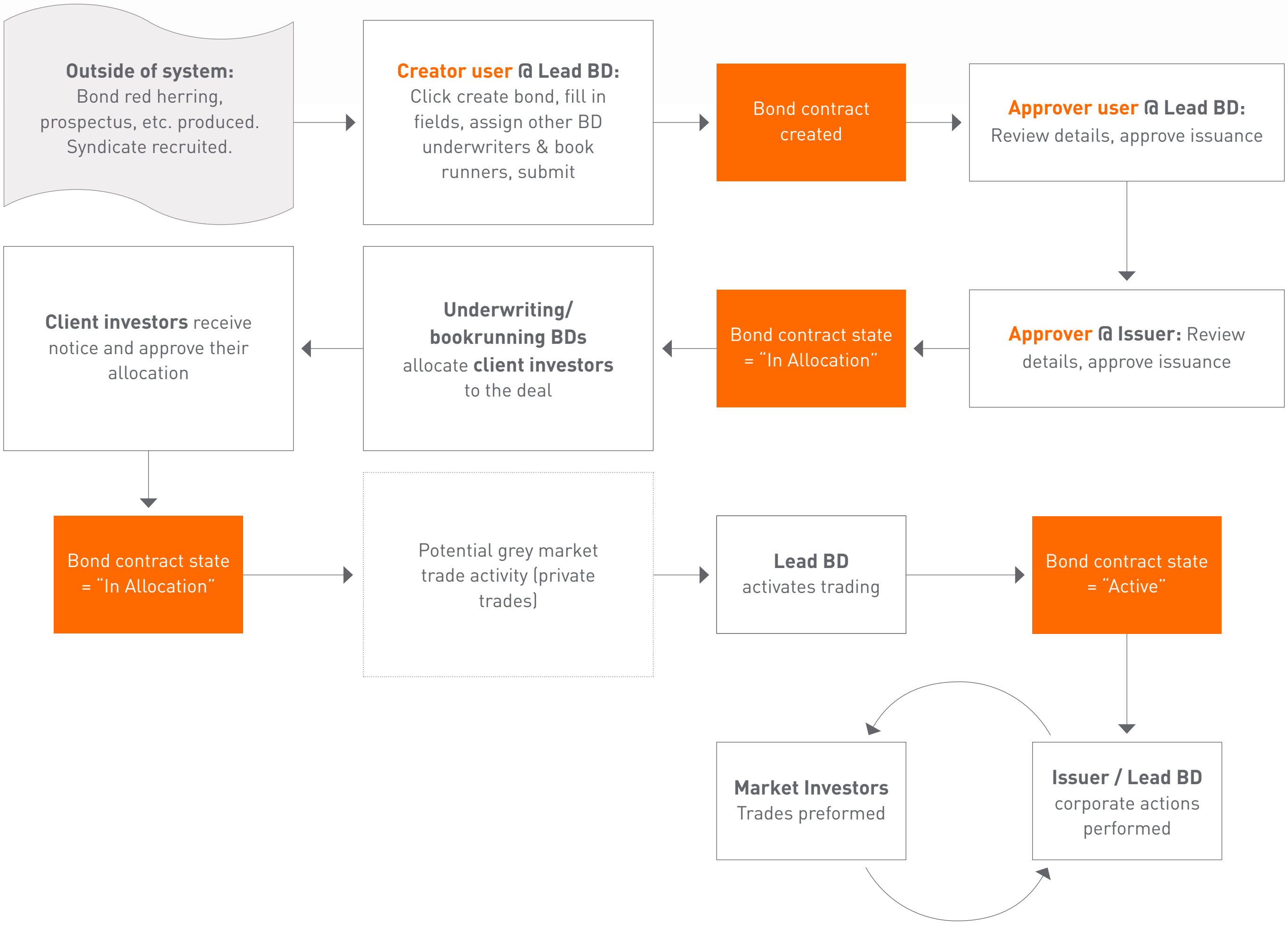
Turing-complete smart contracts are full-fledged computer programs which can be embedded into and “run” on a distributed ledger. By being signed with a private key published into the blockchain, the smart contract code -- and all executions of it made from subsequent transactions -- are fully digitally controlled, and inherit the security of the blockchain itself.

Pseudonyms (i.e. pseudonymous identifiers) embedded in the contract, such as “A-49292-1”, are mapped to public/private key identities and allow control of which parties can do what. And with any use of a smart contract, the code in it can also verify which specific private key is performing the action, and thus can facilitate organizational and role-level access to its logic and data. Contracts can also perform actions at specific times based on a periodic heartbeat, as well as utilize oracles, which are trusted or semi-trusted sources that broadcast a data feed into a distributed ledger, normally at regular intervals. Such oracles can exist for the gold futures spot price or the 5 year Treasury rate.

Benefits of smart contract technology to finance

Turing-complete smart contracts are very powerful tools through which one can essentially specify the functioning of any financial instrument. For instance, like many financial instruments, the lifecycle of a bond can be generally broken into three main components: issuance, primary market allocation, and secondary market trading. While in the secondary market stage, corporate actions may be performed as well.

Today, a majority of corporate bond actions are performed through a significantly manual process consisting of emails, phone calls, faxes, spreadsheet jockeying and so on, all spread across the personnel and systems of numerous counterparties. In contrast, a smart contract representing a corporate bond would capture the properties of the bond such as the involved parties, bond quantity, maturity, coupon, et cetera. Functions would also be defined in the contract that implement bond corporate actions, such as coupon payment, putability/callability and retirement. Human and/or external system access into these processes at the appropriate points could then be facilitated through a user interface that invokes the smart contract behind-the-scenes. This whole application lifecycle would look similar to this:


Outside of system: Bond red herring, prospectus, etc. produced. Syndicate recruited.
Creator user @ Lead BD: Click create bond, fill in fields, assign other BD underwriters & book runners, submit
Bond contract created
Approver user @ Lead BD: Review details, approve issuance
Approver @ Issuer: Review details, approve issuance
Bond contract state = "In Allocation"
Underwriting/ bookrunning BDs allocate client investors to the deal
Client investors receive notice and approve their allocation
Bond contract state = "In Allocation"
Potential grey market trade activity (private trades)
Lead BD activates trading
Bond contract state = "Active"
Issuer / Lead BD corporate actions performed
Market Investors Trades preformed

Some notes on the above:

- All actions would be permissioned at the smart contract level for specific users at specific organizations (counterparties) in the system, based on how the bond was set up.
- Spread pricing could be supported via consultation with the appropriate benchmark rate oracles.
- Cash transfers from one party to another (such as during allocation and secondary market trading of the bond) could utilize gross-settled transfers of an appropriate proxy token (e.g. a USD token), with the sponsoring BDs reconciling token transfers with net-settled cash payments (such as via FedWire or similar) at regular intervals.

The high-level takeaway here is that the smart contract itself functions based on conditions pre-programmed into its code, which are recorded into a distributed ledger that may be jointly operated by the various counterparties. This brings massive efficiency and risk mitigation benefits, as the points of interaction are taken out to a common "rails", with the operations governed by clearly listed, mutually agreed upon terms. With smart contracts, everything is digital, on a single common ledger. Specific counterparties are authenticated as well as authorized to perform actions on the instrument via secure public key cryptography. It's much simpler, safer and more efficient.

Distributed ledgers vs centralized databases

As stated earlier, in today's financial system multiple institutions provide and maintain systems with overlapping functionality. These systems are regularly in conflict, and require slow, expensive and difficult reconciliation. As Richard Gendal Brown explains in his article entitled How to explain the value of replicated, shared ledgers from first principles:

> [The banks themselves have to spend a lot of time and money developing systems that [...] spend even more time and money checking with each other to make sure their systems agree on common facts.

The two primary reasons that these institutions each have their own systems in the first place, instead of a shared system, is to avoid both a single point of failure, and a single point of control. Each institution, or division within an institution, replicates the efforts of other parties to store the state of their clients' and customers' accounts. Again, from Richard Gendal Brown:

> So we have two interesting phenomena: deposit-makers have to trust their banks to be good for the money and to account for things correctly. And the banks themselves have to spend a lot of time and money developing systems that all do pretty much the same thing – and then spend even more time and money checking with each other to make sure their systems agree on common facts.

Each ledger imperfectly replicates the functionality of others. However, the naive solution of centralizing the functionality, and avoiding this replication, reintroduces the single point of failure and the single point of control. The solution is duplication without replication. That is, the data is copied (for redundancy), but the process of copying is not directional. There is no master database which is merely backed up (i.e. replicated); rather, the network is **decentralized**.

Duplication without replication, however, requires decentralized consensus, in which there is no server/client (master/slave) relationship between ledgers, but in which each copy of the ledger is a node in a peer-to-peer system. This consensus system is called **trustless**, because it lacks a single point of control. It is also more robust and resilient than a traditional system, because it is multi-polar and readily scaled.

The authorization and authentication protocols of a decentralized system are fundamentally different from those of a traditional database system, in which there is a gatekeeper party that alone acts on behalf of clients to modify the system state. In a distributed ledger, any party on the network, by contrast, can publish data to the ledger (instant sharing of data), but the effects of those data are limited by the rules of the protocol. Authorization, rather than being a function that is added onto the system at the end, is built in to the lowest level of the stack.

Distributed Ledger	Centraized Database
Consensus on data	Internal and external reconciliation required
Append-only (Immutable)	No restrictions
Distributed	Single point of failure
Decentralized	Single point of control
Peer-to-peer	Unnecessary gateways and middlemen
Cryptographic verification	Cryptography must be added as afterthought
Cryptographic authentication and authorization	Actions are done on behalf of others
Resiliency and availability increase with node count	Backups must be set up 'manually'

Smart contract vs "token"-based systems

Earlier we introduced the concept of a value token, naming the bitcoin token itself as the prototypical example. Bitcoin and other similar value tokens (such as litecoin, dogecoin and so on) act to store and transmit value, as well taking a core role in securing the underlying distributed ledger itself.

We also have explained the concept and purpose of a smart contract. Although the concept of smart contracts has been around for awhile, functioning turing-complete smart contract implementations are still quite novel. Implementations of what we call **token-based systems** preceded these kind of smart contract implementations by several years.

Token-based systems are more primitive than turing-complete smart contract systems. They generally work by allowing a participant in the system to create an arbitrary quantity of some named token, either embedded within a distributed ledger (i.e. via embedded consensus) or utilizing the primitives of one. That token may then be sent amongst the network participants, much like the ledger's native value token is. Tokens can represent a number of things: shares of stock in some company, voting rights, "fuel" to access the feature-set of another software platform, consumption rights to some kind of physical resource (e.g. biofuels), and more.

In the context of modeling financial instruments on a distributed ledger, token-based systems allow one to model quantities but not the actual mechanics of the financial instruments themselves. For instance, let's look at the example of a bond again: For a specific bond issuance, it has a total quantity, which is distributed during a primary market allocation process and changes as the bond trades in the secondary market. It also has "functionality" such as corporate actions (making a coupon payment, puts, calls, conversions, et cetera), and the actual logic behind the issuance, primary market allocation, secondary market trading workflows.

With token-based systems, all of the logic beyond the bond quantities themselves must be implemented outside of the distributed ledger. This means that, with the above example of bonds, one could model simple ownership (i.e. “the owner of address 1234 owns 1,000 of the IBM’s 5 ⅜ of 22 bonds”), but essentially nothing more than that. All of the rest of the bond’s functionality must be implemented within a traditional web/database architecture. Contrast this with smart contract-based technologies, where not only the quantities may be modeled on the distributed ledger, but also all of the logic and workflows themselves, in computer code that all authorized participants may examine and agree on.

Smart Contracts System	Tokens System
Stateful	Stateless
High availability and auditability	Low availability and auditability
All data and logic are stored in the ledger	Only balances or metadata without semantic value are stored on the ledger
May be used to represent the logic of complex financial instruments, including those of corporate actions, payments, dividends, voting, ROFR, restrictions on transfer	May be used to represent only perfectly fungible assets

Contacts @ Symbiont:

Business Inquiries: louis.stone@symbiont.io

Investment Inquiries: mark.smith@symbiont.io

General Information: info@symbiont.io