Corda: A distributed ledger

Mike Hearn

December, 2016

Version 0.2

Confidential: For R3 DLG only - INCOMPLETE

Abstract

A decentralised database with minimal trust between nodes would allow for the creation of a global ledger. Such a ledger would not only be capable of implementing cryptocurrencies but also have many useful applications in finance, trade, supply chain tracking and more. We present Corda, a decentralised global database, and describe in detail how it achieves the goal of providing a robust and easy to use platform for decentralised app development. We elaborate on the high level description provided in the paper *Corda: An introduction* [1] and provide a detailed technical overview, but assume no prior knowledge of the platform.

Contents

1 Introduction **4**

2 Overview **6**

3 The peer to peer network **7**
- 3.1 Network overview . . . 7
- 3.2 Identity and the permissioning service . . . 7
- 3.3 The network map . . . 8
- 3.4 Message delivery . . . 8
- 3.5 Serialization, sessioning, deduplication and signing . . . 9

4 Flow framework **9**
- 4.1 Data visibility and dependency resolution . . . 12

5 Data model **13**
- 5.1 Compound keys . . . 14
- 5.2 Timestamps . . . 15
- 5.3 Attachments and contract bytecodes . . . 16
- 5.4 Hard forks, specifications and dispute resolution . . . 17
- 5.5 Identity lookups . . . 18
- 5.6 Oracles and tear-offs . . . 19
- 5.7 Encumbrances . . . 21
- 5.8 Contract constraints . . . 21

6 Assets and obligations **22**

7 Non-asset instruments **23**

8 Integration with existing infrastructure **23**

9 Deterministic JVM **23**

10 Notaries and consensus **25**
- 10.1 Comparison to Nakamoto block chains . . . 25
- 10.2 Algorithmic agility . . . 26
- 10.3 Validating and non-validating notaries . . . 27
- 10.4 Merging networks . . . 28

11 The vault **29**
- 11.1 Direct SQL access . . . 29

12 Clauses **31**

13 Secure signing devices **31**
- 13.1 Background . . . 31
- 13.2 Confusion attacks . . . 32

13.3 Transaction summaries 33
13.4 Identity substitution 34
13.5 Multi-lingual support 34

14 Client RPC and reactive collections **34**

15 Event scheduling **34**

16 Future work **34**

17 Conclusion **35**

18 Acknowledgements **35**

Bibliography **35**

1 Introduction

In many industries significant effort is needed to keep organisation-specific databases in sync with each other. In the financial sector the effort of keeping different databases synchronised, reconciling them to ensure they actually are synchronised and resolving the 'breaks' that occur when they are not represents a significant fraction of the total work a bank actually does!

Why not just use a shared relational database? This would certainly solve a lot of problems with only existing technology, but it would also raise more questions than answers:

- Who would run this database? Where would we find a sufficient supply of angels to own it?
- In which countries would it be hosted? What would stop that country abusing the mountain of sensitive information it would have?
- What if it got hacked?
- Can you actually scale a relational database to fit the entire financial system within it?
- What happens if The Financial System™ needs to go down for maintenance?
- What kind of nightmarish IT bureaucracy would guard changes to the database schemas?
- How would you manage access control?

We can imagine many other questions. A decentralised database attempts to answer them.

In this paper we differentiate between a *decentralised* database and a *distributed* database. A distributed database like BigTable[2] scales to large datasets and transaction volumes by spreading the data over many computers. However it is assumed that the computers in question are all run by a single homogenous organisation and that the nodes comprising the database all trust each other not to misbehave or leak data. In a decentralised database, such as the one underpinning Bitcoin[3], the nodes make much weaker trust assumptions and actively cross-check each other's work. Such databases trade off performance and usability in order to gain security and global acceptance.

Corda is a decentralised database platform with the following novel features:

- New transaction types can be defined using JVM[4] bytecode.
- Transactions may execute in parallel, on different nodes, without either node being aware of the other's transactions.

- Nodes are arranged in an authenticated peer to peer network. All communication is direct.
- There is no block chain[3]. Transaction races are deconflicted using pluggable *notaries*. A single Corda network may contain multiple notaries that provide their guarantees using a variety of different algorithms. Thus Corda is not tied to any particular consensus algorithm. (§10)
- Data is shared on a need-to-know basis. Nodes provide the dependency graph of a transaction they are sending to another node on demand, but there is no global broadcast of *all* transactions.
- Bytecode-to-bytecode transpilation is used to allow complex, multi-step transaction building protocols called *flows* to be modelled as blocking code. The code is transformed into an asynchronous state machine, with checkpoints written to the node's backing database when messages are sent and received. A node may potentially have millions of flows active at once and they may last days, across node restarts and even upgrades. Flows expose progress information to node administrators and users and may interact with people as well as other nodes.
- The data model allows for arbitrary object graphs to be stored in the ledger. These graphs are called *states* and are the atomic unit of data.
- Nodes are backed by a relational database and data placed in the ledger can be queried using SQL as well as joined with private tables, thanks to slots in the state definitions that are reserved for join keys.
- The platform provides a rich type system for the representation of things like dates, currencies, legal entities and so on.
- States can declare a relational mapping and can be queried using SQL.
- Integration with existing systems is considered from the start. The network can support rapid bulk data imports from other database systems without placing load on the network. Events on the ledger are exposed via an embedded JMS compatible message broker.
- States can declare scheduled events. For example a bond state may declare an automatic transition to a "in default" state if it is not repaid in time.

Corda follows a general philosophy of reusing existing proven software systems and infrastructure where possible. Comparisons with Bitcoin and Ethereum will be provided throughout.

2 Overview

Corda is a platform for the writing of "CorDapps": applications that extend the global database with new capabilities. Such apps define new data types, new inter-node protocol flows and the "smart contracts" that determine allowed changes.

What is a smart contract? That depends on the model of computation we are talking about. There are two competing computational models used in decentralised databases: the virtual computer model and the UTXO model. The virtual computer model is used by Ethereum [5]. It models the database as the in-memory state of a global computer with a single thread of execution determined by the block chain. In the UTXO model, as used in Bitcoin, the database is a set of immutable rows keyed by `(hash:output index)`. Transactions define outputs that append new rows and inputs which consume existing rows. The term "smart contract" has a different meaning in each model. A deeper discussion of the tradeoffs and terminology in the different approaches can be found in the Corda introductory paper [1].

We use the UTXO model and as a result our transactions are structurally similar to Bitcoin transactions: they have inputs, outputs and signatures. Unlike Bitcoin, Corda database rows can contain arbitrary data, not just a value field. Because the data consumed and added by transactions is not necessarily a set of key/value pairs, we don't talk about rows but rather *states*. Like Bitcoin, Corda states are associated with bytecode programs that must accept a transaction for it to be valid, but unlike Bitcoin, a transaction must satisfy the programs for both the input and output states at once. *Issuance transactions* may append new states to the database without consuming any existing states but unlike in Bitcoin these transactions are not special and may be created at any time, by anyone.

In contrast to both Bitcoin and Ethereum, Corda does not order transactions using a block chain and by implication does not use miners or proof-of-work. Instead each state points to a *notary*, which is a service that guarantees it will sign a transaction only if all the input states are un-consumed. A transaction is not allowed to consume states controlled by multiple notaries and thus there is never any need for two-phase commit between notaries. If a combination of states would cross notaries then a special transaction type is used to move them onto a single notary first. See §10 for more information.

The Corda transaction format has various other features which are described in later sections.

3 The peer to peer network

3.1 Network overview

A Corda network consists of the following components:

- Nodes, communicating using AMQP/1.0 over TLS. Nodes use a relational database for data storage.
- A permissioning service that automates the process of provisioning TLS certificates.
- A network map service that publishes information about nodes on the network.
- One or more notary services. A notary may itself be distributed over multiple nodes.
- Zero or more oracle services. An oracle is a well known service that signs transactions if they state a fact and that fact is considered to be true. They may also optionally also provide the facts. This is how the ledger can be connected to the real world, despite being fully deterministic.

A purely in-memory implementation of the messaging subsystem is provided which can inject simulated latency between nodes and visualise communications between them. This can be useful for debugging, testing and educational purposes.

Oracles and notaries are covered in later sections.

3.2 Identity and the permissioning service

Unlike Bitcoin and Ethereum, Corda is designed for semi-private networks in which admission requires obtaining an identity signed by a root authority. This assumption is pervasive - the flow API provides messaging in terms of identities, with routing and delivery to underlying nodes being handled automatically. There is no global broadcast at any point.

This 'identity' does not have to be a legal or true identity. In the same way that an email address is a globally unique pseudonym that is ultimately rooted by the top of the DNS hierarchy, so too can a Corda network work with arbitrary self-selected usernames. The permissioning service can implement any policy it likes as long as the identities it signs are globally unique. Thus an entirely anonymous Corda network is possible if a suitable IP obfuscation system like Tor is also used.

Whilst simple string identities are likely sufficient for some networks, the financial industry typically requires some level of *know your customer* checking, and differentiation between different legal entities, branches and desks that may

share the same brand name. Corda reuses the standard PKIX infrastructure for connecting public keys to identities and thus names are actually X.500 names. When a single string is sufficient the *common name* field can be used alone, similar to the web PKI. In more complex deployments the additional structure X.500 provides may be useful to differentiate between entities with the same name. For example there are at least five different companies called *American Savings Bank* and in the past there may have been more than 40 independent banks with that name.

More complex notions of identity that may attest to many time-varying attributes are not handled at this layer of the system: the base identity is always just an X.500 name. Note that even though messaging is always identified, transactions themselves may still contain anonymous public keys.

3.3 The network map

Every network requires a network map service, which may itself be composed of multiple cooperating nodes. This is similar to Tor's concept of *directory authorities*. The network map publishes the IP addresses through which every node on the network can be reached, along with the identity certificates of those nodes and the services they provide. On receiving a connection, nodes check that the connecting node is in the network map.

The network map abstracts the underlying IP addresses of the nodes from more useful business concepts like identities and services. Each participant on the network, called a *party*, publishes one or more IP addresses in the network map. Equivalent domain names may be helpful for debugging but are not required. User interfaces and APIs always work in terms of identities - there is thus no equivalent to Bitcoin's notion of an address (hashed public key), and user-facing applications rely on auto-completion and search rather than QRcodes to identify a logical recipient.

It is possible to subscribe to network map changes and registering with the map is the first thing a node does at startup. Nodes may optionally advertise their nearest city for load balancing and network visualisation purposes.

The map is a document that may be cached and distributed throughout the network. The map is therefore not required to be highly available: if the map service becomes unreachable new nodes may not join the network and existing nodes may not change their advertised service set, but otherwise things continue as normal.

3.4 Message delivery

The network is structurally similar to the email network. Nodes are expected to be long lived but may depart temporarily due to crashes, connectivity interrup-

tions or maintenance. Messages are written to disk and delivery is retried until the remote node has acknowledged a message, at which point it is expected to have either reliably stored the message or processed it completely. Connections between nodes are built and torn down as needed: there is no assumption of constant connectivity. An ideal network would be entirely flat with high quality connectivity between all nodes, but Corda recognises that this is not always compatible with common network setups and thus the message routing component of a node can be separated from the rest and run outside the firewall. In this way nodes that do not have duplex connectivity can still take part in the network as first class citizens. Additionally a single node may have multiple advertised IP addresses.

The reference implementation provides this functionality using the Apache Artemis message broker, through which it obtains journalling, load balancing, flow control, high availability clustering, streaming of messages too large to fit in RAM and many other useful features. The network uses the *AMQP/1.0*[8] protocol which is a widely implemented binary messaging standard, combined with TLS to secure messages in transit and authenticate the endpoints.

3.5 Serialization, sessioning, deduplication and signing

All messages are encoded using a compact binary format. Each message has a UUID set in an AMQP header which is used as a deduplication key, thus accidentally redelivered messages will be ignored.

Messages may also have an associated organising 64-bit *session ID*. Note that this is distinct from the AMQP notion of a session. Sessions can be long lived and persist across node restarts and network outages. They exist in order to group messages that are part of a *flow*, described in more detail below.

Messages that are successfully processed by a node generate a signed acknowledgement message called a 'receipt'. Note that this is distinct from the unsigned acknowledgements that live at the AMQP level and which simply flag that a message was successfully downloaded over the wire. A receipt may be generated some time after the message is processed in the case where acknowledgements are being batched to amortise signing overhead, and the receipt identifies the message by the hash of its content. The purpose of the receipts is to give a node undeniable evidence that a counterparty received a notification that would stand up later in a dispute mediation process. Corda does not attempt to support deniable messaging.

4 Flow framework

It is common in decentralised ledger systems for complex multi-party protocols to be needed. The Bitcoin payment channel protocol[9] involves two parties

putting money into a multi-signature pot, then iterating with your counterparty a shared transaction that spends that pot, with extra transactions used for the case where one party or the other fails to terminate properly. Such protocols typically involve reliable private message passing, checkpointing to disk, signing of transactions, interaction with the p2p network, reporting progress to the user, maintaining a complex state machine with timeouts and error cases, and possibly interaction with internal systems on either side. All this can become quite involved. The implementation of Bitcoin payment channels in the bitcoinj library is approximately 9000 lines of Java, very little of which involves cryptography.

As another example, the core Bitcoin protocol only allows you to append transactions to the ledger. Transmitting other information that might be useful such as a text message, refund address, identity information and so on is not supported and must be handled in some other way - typically by wrapping the raw ledger transaction bytes in a larger message that adds the desired metadata and giving responsibility for broadcasting the embedded transaction to the recipient, as in Bitcoin's BIP 70[10].

In Corda transaction data is not globally broadcast. Instead it is transmitted to the relevant parties only when they need to see it. Moreover even quite simple use cases - like sending cash - may involve a multi-step negotiation between counterparties and the involvement of a third party such as a notary. Additional information that isn't put into the ledger is considered essential, as opposed to nice-to-have. Thus unlike traditional blockchain systems in which the primary form of communication is global broadcast, in Corda *all* communication takes the form of small multi-party sub-protocols called flows.

The flow framework presents a programming model that looks to the developer as if they have the ability to run millions of long lived threads which can survive node restarts, and even node upgrades. APIs are provided to send and receive object graphs to and from other identities on the network, embed sub-flows, and report progress to observers. In this way business logic can be expressed at a very high level, with the details of making it reliable and efficient abstracted away. This is achieved with the following components.

Just-in-time state machine compiler. Code that is written in a blocking manner typically cannot be stopped and transparently restarted later. The first time a flow's `call` method is invoked a bytecode-to-bytecode transformation occurs that rewrites the classes into a form that implements a resumable state machine. These state machines are sometimes called fibers or coroutines, and the transformation engine Corda uses is capable of rewriting code arbitrarily deep in the stack on the fly. The developer may thus break his or her logic into multiple methods and classes, use loops, and generally structure their program as if it were executing in a single blocking thread. There's only a small list of things they should not do: sleeping, directly accessing the network APIs, or

doing other tasks that might block outside of the framework.

Transparent checkpointing. When a flow wishes to wait for a message from another party (or input from a human being) the underlying stack frames are suspended onto the heap, then crawled and serialized into the node's underlying relational database using an object serialization framework. The written objects are prefixed with small schema definitions that allow some measure of portability across changes to the layout of objects, although portability across changes to the stack layout is left for future work. Flows are resumed and suspended on demand, meaning it is feasible to have far more flows active at once than would fit in memory. The checkpointing process is atomic with changes to local storage and acknowledgement of network messages.

Identity to IP address mapping. Flows are written in terms of identities. The framework takes care of routing messages to the right IP address for a given identity, following movements that may take place whilst the flow is active and handling load balancing for multi-homed parties as appropriate.

A library of subflows. Flows can invoke sub-flows, and a library of flows is provided to automate common tasks like notarising a transaction or atomically swapping ownership of two assets.

Progress reporting. Flows can provide a progress tracker that indicates which step they are up to. Steps can have human-meaningful labels, along with other tagged data like a progress bar. Progress trackers are hierarchical and steps can have sub-trackers for invoked sub-flows.

Flow hospital. Flows can pause if they throw exceptions or explicitly request human assistance. A flow that has stopped appears in the *flow hospital* where the node's administrator may decide to kill the flow or provide it with a solution. The ability to request manual solutions is useful for cases where the other side isn't sure why you are contacting them, for example, the specified reason for sending a payment is not recognised, or when the asset used for a payment is not considered acceptable.

Flows are named using reverse DNS notation and several are defined by the base protocol. Note that the framework is not required to implement the wire protocols, it is just a development aid.

4.1 Data visibility and dependency resolution

When a transaction is presented to a node as part of a flow it may need to be checked. Simply sending you a message saying that I am paying you £1000 is only useful if youa are sure I own the money I'm using to pay me. Checking transaction validity is the responsibility of the `ResolveTransactions` flow. This flow performs a breadth-first search over the transaction graph, downloading any missing transactions into local storage and validating them. The search bottoms out at the issuance transactions. A transaction is not considered valid if any of its transitive dependencies are invalid.

It is required that a node be able to present the entire dependency graph for a transaction it is asking another node to accept. Thus there is never any confusion about where to find transaction data. Because transactions are always communicated inside a flow, and flows embed the resolution flow, the necessary dependencies are fetched and checked automatically from the correct peer. Transactions propagate around the network lazily and there is no need for distributed hash tables.

This approach has several consequences. One is that transactions that move highly liquid assets like cash may end up becoming a part of a very long chain of transactions. The act of resolving the tip of such a graph can involve many round-trips and thus take some time to fully complete. How quickly a Corda network can send payments is thus difficult to characterise: it depends heavily on usage and distance between nodes. Whilst nodes could pre-push transactions in anticipation of them being fetched anyway, such optimisations are left for future work.

A more important consequence is that in the absence of additional privacy measures it is difficult to reason about who may get to see transaction data. We can say it's definitely better than a system that uses global broadcast, but how much better is hard to characterise. This uncertainty is mitigated by several factors.

Small-subgraph transactions. Some uses of the ledger do not involve widely circulated asset states. For example, two institutions that wish to keep their view of a particular deal synchronised but who are making related payments off-ledger may use transactions that never go outside the involved parties. A discussion of on-ledger vs off-ledger cash can be found in a later section.

Transaction privacy techniques. Corda supports a variety of transaction data hiding techniques. For example, public keys can be randomised to make it difficult to link transactions to an identity. "Tear-offs" (§5.6) allow some parts of a transaction to be presented without the others. In future versions of the system secure hardware and/or zero knowledge proofs could be used to

convince a party of the validity of a transaction without revealing the underlying data.

State re-issuance. In cases where a state represents an asset that is backed by a particular issuer, and the issuer is trusted to behave atomically even when the ledger isn't forcing atomicity, the state can simply be 'exited' from the ledger and then re-issued. Because there are no links between the exit and reissue transactions this shortens the chain. In practice most issuers of highly liquid assets are already trusted with far more sensitive tasks than reliably issuing pairs of signed data structures, so this approach is unlikely to be an issue.

5 Data model

Transactions consist of the following components:

Input references — These are `(hash, output index)` pairs that point to the states a transaction is consuming.

Output states — Each state specifies the notary for the new state, the contract(s) that define its allowed transition functions and finally the data itself.

Attachments — Transactions specify an ordered list of zip file hashes. Each zip file may contain code, data, certificates or supporting documentation for the transaction. Contract code has access to the contents of the attachments when checking the transaction for validity.

Commands — There may be multiple allowed output states from any given input state. For instance an asset can be moved to a new owner on the ledger, or issued, or exited from the ledger if the asset has been redeemed by the owner and no longer needs to be tracked. A command is essentially a parameter to the contract that specifies more information than is obtainable from examination of the states by themselves (e.g. data from an oracle service). Each command has an associated list of public keys. Like states, commands are object graphs.

Signatures — The set of required signatures is equal to the union of the commands' public keys.

Type — Transactions can either be normal or notary-changing. The validation rules for each are different.

Timestamp — When present, a timestamp defines a time range in which the transaction is considered to have occurrred. This is discussed in more detail below.

Summaries — Textual summaries of what the transaction does, checked by the involved smart contracts. This field is useful for secure signing devices (see §13).

Signatures are appended to the end of a transaction and transactions are identified by the hash used for signing, so signature malleability is not a problem. There is never a need to identify a transaction including its accompanying signatures by hash. Signatures can be both checked and generated in parallel, and they are not directly exposed to contract code. Instead contracts check that the set of public keys specified by a command is appropriate, knowing that the transaction will not be valid unless every key listed in every command has a matching signature. Public key structures are themselves opaque. In this way algorithmic agility is retained: new signature algorithms can be deployed without adjusting the code of the smart contracts themselves.

5.1 Compound keys

The term "public key" in the description above actually refers to a *compound key*. Compound keys are trees in which leaves are regular cryptographic public keys with an accompanying algorithm identifiers. Nodes in the tree specify both the weights of each child and a threshold weight that must be met. The validty of a set of signatures can be determined by walking the tree bottom-up, summing the weights of the keys that have a valid signature and comparing against the threshold. By using weights and thresholds a variety of conditions can be encoded, including boolean formulas with AND and OR.

Compound keys are useful in multiple scenarios. For example, assets can be placed under the control of a 2-of-2 compound key where one leaf key is owned by a user, and the other by an independent risk analysis system. The risk analysis system refuses to sign if the transaction seems suspicious, like if too much value has been transferred in too short a time window. Another example involves encoding corporate structures into the key, allowing a CFO to sign a large transaction alone but his subordinates are required to work together. Compound keys are also useful for notaries. Each participant in a distributed notary is represented by a leaf, and the threshold is set such that some participants can be offline or refusing to sign yet the signature of the group is still valid.

Whilst there are threshold signature schemes in the literature that allow compound keys and signatures to be produced mathematically, we choose the less space efficient explicit form in order to allow a mixture of keys using different algorithms. In this way old algorithms can be phased out and new algorithms phased in without requiring all participants in a group to upgrade simultaneously.

5.2 Timestamps

Transaction timestamps specify a `[start, end]` time window within which the transaction is asserted to have occurred. Timestamps are expressed as windows because in a distributed system there is no true time, only a large number of desynchronised clocks. This is not only implied by the laws of physics but also by the nature of shared transactions - especially if the signing of a transaction requires multiple human authorisations, the process of constructing a joint transaction could take hours or even days.

It is important to note that the purpose of a transaction timestamp is to communicate the transaction's position on the timeline to the smart contract code for the enforcement of contractual logic. Whilst such timestamps may also be used for other purposes, such as regulatory reporting or ordering of events in a user interface, there is no requirement to use them like that and locally observed timestamps may sometimes be preferable even if they will not exactly match the time observed by other parties. Alternatively if a precise point on the timeline is required and it must also be agreed by multiple parties, the midpoint of the time window may be used by convention. Even though this may not precisely align to any particular action (like a keystroke or verbal agreement) it is often useful nonetheless.

Timestamp windows may be open ended in order to communicate that the transaction occurred before a certain time or after a certain time, but how much before or after is unimportant. This can be used in a similar way to Bitcoin's `nLockTime` transaction field, which specifies a *happens-after* constraint.

Timestamps are checked and enforced by notary services. As the participants in a notary service will themselves not have precisely aligned clocks, whether a transaction is considered valid or not at the moment it is submitted to a notary may be unpredictable if submission occurs right on a boundary of the given window. However, from the perspective of all other observers the notaries signature is decisive: if the signature is present, the transaction is assumed to have occurred within that time.

Reference clocks. In order to allow for relatively tight time windows to be used when transactions are fully under the control of a single party, notaries are expected to be synchronised to the atomic clocks at the US Naval Observatory. Accurate feeds of this clock can be obtained from GPS satellites. Note that Corda uses the Java timeline[11] which is UTC with leap seconds spread over the last 1000 seconds of the day, thus each day always has exactly 86400 seconds. Care should be taken to ensure that changes in the GPS leap second counter are correctly smeared in order to stay synchronised with Java time. When setting a transaction time window care must be taken to account for network propagation delays between the user and the notary service, and messaging within the notary service.

5.3 Attachments and contract bytecodes

Transactions may have a number of *attachments*, identified by the hash of the file. Attachments are stored and transmitted separately to transaction data and are fetched by the standard resolution flow only when the attachment has not previously been seen before.

Attachments are always zip files[12] and cannot be referred to individually by contract code. The files within the zips are collapsed together into a single logical file system, with overlapping files being resolved in favour of the first mentioned. Not coincidentally, this is the mechanism used by Java classpaths.

Smart contracts in Corda are defined using JVM bytecode as specified in *"The Java Virtual Machine Specification SE 8 Edition"*[4], with some small differences that are described in a later section. A contract is simply a class that implements the `Contract` interface, which in turn exposes a single function called `verify`. The verify function is passed a transaction and either throws an exception if the transaction is considered to be invalid, or returns with no result if the transaction is valid. The set of verify functions to use is the union of the contracts specified by each state (which may be expressed as constraints, see §5.8). Embedding the JVM specification in the Corda specification enables developers to write code in a variety of languages, use well developed toolchains, and to reuse code already authored in Java or other JVM compatible languages.

The Java standards also specify a comprehensive type system for expressing common business data. Time and calendar handling is provided by an implementation of the JSR 310 specification, decimal calculations can be performed either using portable ('`strictfp`') floating point arithmetic or the provided bignum library, and so on. These libraries have been carefully engineered by the business Java community over a period of many years and it makes sense to build on this investment.

Contract bytecode also defines the states themselves, which may be arbitrary object graphs. Because JVM classes are not a convenient form to work with from non-JVM platforms the allowed types are restricted and a standardised binary encoding scheme is provided. States may label their properties with a small set of standardised annotations. These can be useful for controlling how states are serialised to JSON and XML (using JSR 367 and JSR 222 respectively), for expressing static validation constraints (JSR 349) and for controlling how states are inserted into relational databases (JSR 338). This feature is discussed later.

Attachments may also contain data files that support the contract code. These may be in the same zip as the bytecode files, or in a different zip that must be provided for the transaction to be valid. Examples of such data files might include currency definitions, timezone data and public holiday calendars. Any public data may be referenced in this way. Attachments are intended for data on the ledger that many parties may wish to reuse over and over again. Data files

are accessed by contract code using the same APIs as any file on the classpath would be accessed. The platform imposes some restrictions on what kinds of data can be included in attachments along with size limits, to avoid people placing inappropriate files on the global ledger (videos, PowerPoints etc).

Note that the creator of a transaction gets to choose which files are attached. Therefore, it is typical that states place constraints on the data they're willing to accept. Attachments *provide* data but do not *authenticate* it, so if there's a risk of someone providing bad data to gain an economic advantage there must be a constraints mechanism to prevent that from happening. This is rooted at the contract constraints encoded in the states themselves: a state can not only name a class that implements the `Contract` interface but also place constraints on the zip/jar file that provides it. That constraint can in turn be used to ensure that the contract checks the authenticity of the data - either by checking the hash of the data directly, or by requiring the data to be signed by some trusted third party.

5.4 Hard forks, specifications and dispute resolution

Decentralised ledger systems often differ in their underlying political ideology as well as their technical choices. The Ethereum project originally promised "unstoppable apps" which would implement "code as law". After a prominent smart contract was hacked, an argument took place over whether what had occurred could be described as a hack at all given the lack of any non-code specification of what the program was meant to do. The disagreement eventually led to a split in the community.

As Corda contracts are simply zip files, it is easy to include a PDF or other documents describing what a contract is meant to actually do. There is no requirement to use this mechanism, and there is no requirement that these documents have any legal weight. However in financial use cases it's expected that they would be legal contracts that take precedence over the software implementations in case of disagreement.

It is technically possible to write a contract that cannot be upgraded. If such a contract governed an asset that existed only on the ledger, like a cryptocurrency, then that would provide an approximation of "code as law". We leave discussion of this wisdom of this concept to political scientists and reddit.

Platform logging There is no direct equivalent in Corda of a block chain "hard fork", so the only solution to discarding buggy or fraudulent transaction chains would be to mutually agree out of band to discard an entire transaction subgraph. As there is no global visibility either this mutual agreement would not need to encompass all network participants: only those who may have received and processed such transactions. The flip side of lacking global visibility is that there is no single point that records who exactly has seen which transactions.

Determining the set of entities that'd have to agree to discard a subgraph means correlating node activity logs. Corda nodes log sufficient information to ensure this correlation can take place. The platform defines a flow to assist with this, which can be used by anyone. A tool is provided that generates an "investigation request" and sends it to a seed node. The flow signals to the node administrator that a decision is required, and sufficient information is transmitted to the node to try and convince the administrator to take part (e.g. a signed court order). If the administrator accepts the request through the node explorer interface, the next hops in the transaction chain are returned. In this way the tool can semi-automatically crawl the network to find all parties that would be affected by a proposed rollback. The platform does not take a position on what types of transaction rollback are justified and provides only minimal support for implementing rollbacks beyond locating the parties that would have to agree.

Once involved parties are identified there are at least two strategies for editing the ledger. One is to extend the transaction chain with new transactions that simply correct the database to match the intended reality. For this to be possible the smart contract must have been written to allow arbitrary changes outside its normal business logic when a sufficient threshold of signatures is present. This strategy is simple and makes the most sense when the number of parties involved in a state is small and parties have no incentive to leave bad information in the ledger. For asset states that are the result of theft or fraud the only party involved in a state may resist attempts to patch things up in this way, as they may be able to benefit in the real world from the time lag between the ledger becoming inaccurate and it catching up with reality. In this case a more complex approach can be used in which the involved parties minus the uncooperative party agree to mark the relevant states as no longer consumed/spent. This is essentially a limited form of database rollback.

5.5 Identity lookups

In all block chain inspired systems there exists a tension between wanting to know who you are dealing with and not wanting others to know. A standard technique is to use randomised public keys in the shared data, and keep the knowledge of the identity that key maps to private. For instance, it is considered good practice to generate a fresh key for every received payment. This technique exploits the fact that verifying the integrity of the ledger does not require knowing exactly who took part in the transactions, only that they followed the agreed upon rules of the system.

Platforms such as Bitcoin and Ethereum have relatively ad-hoc mechanisms for linking identities and keys. Typically it is the user's responsibility to manually label public keys in their wallet software using knowledge gleaned from websites, shop signs and so on. Because these mechanisms are ad hoc and tedious many users don't bother, which can make it hard to figure out where money

went later. It also complicates the deployment of secure signing devices and risk analysis engines. Bitcoin has BIP 70[10] which specifies a way of signed a "payment request" using X.509 certificates linked to the web PKI, giving a cryptographically secured and standardised way of knowing who you are dealing with. Identities in this system are the same as used in the web PKI: a domain name, email address or EV (extended validation) organisation name.

Corda takes this concept further. States may define fields of type `Party`, which encapsulates an identity and a public key. When a state is deserialised from a transaction in its raw form, the identity field of the `Party` object is null and only the public (compound) key is present. If a transaction is deserialised in conjunction with X.509 certificate chains linking the transient public keys to long term identity keys the identity field is set. In this way a single data representation can be used for both the anonymised case, such as when validating dependencies of a transaction, and the identified case, such as when trading directly with a counterparty. Trading flows incorporate sub-flows to transmit certificates for the keys used, which are then stored in the local database. However the transaction resolution flow does not transmit such data, keeping the transactions in the chain of custody pseudonymous.

Deterministic key derivation Corda allows for but does not mandate the use of determinstic key derivation schemes such as BIP 32[13]. The infrastructure does not assume any mathematical relationship between public keys because some cryptographic schemes are not compatible with such systems. Thus we take the efficiency hit of always linking transient public keys to longer term keys with X.509 certificates.

5.6 Oracles and tear-offs

It is sometimes convenient to reveal a small part of a transaction to a counterparty in a way that allows them to check the signatures and sign it themselves. A typical use case for this is an *oracle*, defined as a network service that is trusted to sign transactions containing statements about the world outside the ledger only if the statements are true.

Here are some example statements an oracle might check:

- The price of a stock at a particular moment was X.
- An agreed upon interest rate at a particular moment was Y.
- If a specific organisation has declared bankruptcy.
- Weather conditions in a particular place at a particular time.

It is worth asking why a smart contract cannot simply fetch this information from some internet server itself: why do we insist on this notion of an oracle. The reason is that all calculations on the ledger must be deterministic. Everyone

must be able to check the validity of a transaction and arrive at exactly the same answer, at any time (including years into the future), on any kind of computer. If a smart contract could do things like read the system clock or fetch arbitrary web pages then it would be possible for some computers to conclude a transaction was valid, whilst others concluded it was not (e.g. if the remote server had gone offline). Solving this problem means all the data needed to check the transaction must be in the ledger, which in turn implies that we must accept the point of view of some specific observer. That way there can be no disagreement about what happened.

One way to implement oracles would be to have them sign a small data structure which is then embedded somewhere in a transaction (in a state or command). We take a different approach in which oracles sign the entire transaction, and data the oracle doesn't need to see is "torn off" before the transaction is sent. This is done by structuring the transaction as a Merkle hash tree so that the hash used for the signing operation is the root. By presenting a counterparty with the data elements that are needed along with the Merkle branches linking them to the root hash, that counterparty can sign the entire transaction whilst only being able to see some of it. Additionally, if the counterparty needs to be convinced that some third party has already signed the transaction, that is also straightforward. Typically an oracle will be presented with the Merkle branches for the command or state that contains the data, and the timestamp field, and nothing else. The resulting signature contains flag bits indicating which parts of the structure were presented for signing to avoid a single signature covering more than expected.

There are a couple of reasons to take this more indirect approach. One is to keep a single signature checking code path. By ensuring there is only one place in a transaction where signatures may be found, algorithmic agility and parallel/batch verification are easy to implement. When a signature may be found in any arbitrary location in a transaction's data structures, and where verification may be controlled by the contract code itself (as in Bitcoin), it becomes harder to maximise signature checking efficiency. As signature checks are often one of the slowest parts of a block chain system, it is desirable to preserve these capabilities.

Another reason is to provide oracles with a business model. If oracles just signed statements and nothing else then it would be difficult to run an oracle in which there are only a small number of potential statements, but determining their truth is very expensive. People could share the signed statements and reuse them in many different transactions, meaning the cost of issuing the initial signatures would have to be very high, perhaps unworkably high. Because oracles sign specific transactions, not specific statements, an oracle that is charging for its services can amortise the cost of determining the truth of a statement over many users who cannot then share the signature itself (because it covers a one-time-use structure by definition).

5.7 Encumbrances

Each state in a transaction specifies a contract (boolean function) that is invoked with the entire transaction as input. All contracts must accept in order for the transaction to be considered valid. Sometimes we would like to compose the behaviours of multiple different contracts. Consider the notion of a "time lock" - a restriction on a state that prevents it being modified (i.e. sold) until a certain time. This is a general piece of logic that could apply to many kinds of assets. Whilst such logic could be implemented in a library and then called from every contract that might want to benefit from it, that requires all contract authors to think ahead and include the functionality. It would be better if we could mandate that the time lock logic ran along side the contract that governs the locked state.

Consider an asset that is supposed to remain frozen until a time is reached. Encumbrances allow a state to specify another state that must be present in any transaction that consumes it. For example, a time lock contract can define a state that contains the time at which the lock expires, and a simple contract that just compares that time against the transaction timestamp. The asset state can be included in a spend-to-self transaction that doesn't change the ownership of the asset but does include a time lock state in the outputs. Now if the asset state is used, the time lock state must also be used, and that triggers the execution of the time lock contract.

Encumbered states can only point to one encumbrance state, but that state can itself point to another and so on, resulting in a chain of encumbrances all of which must be satisfied.

An encumbrance state must be present in the same transaction as the encumbered state, as states refer to each other by index alone.

5.8 Contract constraints

The simplest way of tying states to the contract code that defines them is by hash. This works for very simple and stable programs, but more complicated contracts may need to be upgraded. In this case it may be preferable for states to refer to contracts by the identity of the signer. Because contracts are stored in zip files, and because a Java Archive (JAR) file is just a zip with some extra files inside, it is possible to use the standard JAR signing infrastructure to identify the source of contract code. Simple constraints such as "any contract of this name signed by these keys" allow for some upgrade flexibility, at the cost of increased exposure to rogue contract developers. Requiring combinations of signatures helps reduce the risk of a rogue or hacked developer publishing a bad contract version, at the cost of increased difficulty in releasing new versions. State creators may also specify third parties they wish to review contract code.

Regardless of which set of tradeoffs is chosen, the framework can accomodate them.

A contract constraint may use a compound key of the type described in §5.1. The standard JAR signing protocol allows for multiple signatures from different private keys, thus being able to satisfy compound keys. The allowed signing algorithms are `SHA256withRSA` and `SHA256withECDSA`. Note that the cryptographic algorithms used for code signing may not always be the same as those used for transaction signing, as for code signing we place initial focus on being able to re-use the infrastructure.

6 Assets and obligations

A ledger that cannot record the ownership of assets is not very useful. We define a set of classes that model asset-like behaviour and provide some platform contracts to ensure interoperable notions of cash and obligations.

We define the notion of an `OwnableState`, implemented as an interface which any state may conform to. Ownable states are required to have an `owner` field which is a compound key (see §5.1). This is utilised by generic code in the vault (see §11) to manipulate ownable states.

From `OwnableState` we derive a `FungibleAsset` concept to represent assets of measurable quantity, in which units are sufficiently similar to represented together in a single ledger state. Making that concrete, pound notes are a fungible asset: regardless of whether you represent £10 as a single £10 note or two notes of £5 each the total value is the same. Other kinds of fungible asset could be barrels of Brent Oil (but not all kinds of crude oil worldwide, because oil comes in different grades which are not interchangeable), litres of clean water, kilograms of bananas, units of a stock and so on.

When cash is represented on a digital ledger an additional complication can arise: for national "fiat" currencies the ledger merely records an entity that has a liability which may be redeemed for some other form (physical currency, a wire transfer via some other ledger system, etc). This means that two ledger entries of £1000 may *not* be entirely fungible because all the entries really represent is a claim on an issuer, which - if it is not a central bank - may go bankrupt. Even assuming defaults never happen, the data representing where an asset may be redeemed must be tracked through the chain of custody, so 'exiting' the asset from the ledger and thus claiming physical ownership can be done.

The Corda type system supports the encoding of this complexity. The `Amount<T>` type defines an integer quantity of some token. This type does not support fractional quantities so when used to represent national currencies the quantity must be measured in pennies, with sub-penny amount requiring the use of some other type. The token can be represented by any type. A common token type to use

is `Issued<T>`, which defines a token issued by some party. It encapsulates what the asset is, who issued it, and an opaque reference field that is not parsed by the platform - it is intended to help the issuer keep track of e.g. an account number, the location where the asset can be found in storage, etc.

TODO: Complete this section

7 Non-asset instruments

TODO

8 Integration with existing infrastructure

TODO

9 Deterministic JVM

It is important that all nodes that process a transaction always agree on whether it is valid or not. Because transaction types are defined using JVM bytecode, this means the execution of that bytecode must be fully deterministic. Out of the box a standard JVM is not fully deterministic, thus we must make some modifications in order to satisfy our requirements. Non-determinism could come from the following sources:

- Sources of external input e.g. the file system, network, system properties, clocks.
- Random number generators.
- Different decisions about when to terminate long running programs.
- `Object.hashCode()`, which is typically implemented either by returning a pointer address or by assigning the object a random number. This can surface as different iteration orders over hash maps and hash sets.
- Differences in hardware floating point arithmetic.
- Multi-threading.
- Differences in API implementations between nodes.
- Garbage collector callbacks.

To ensure that the contract verify function is fully pure even in the face of infinite loops we construct a new type of JVM sandbox. It utilises a bytecode static analysis and rewriting pass, along with a small JVM patch that allows

the sandbox to control the behaviour of hashcode generation. Contract code is rewritten the first time it needs to be executed and then stored for future use.

The bytecode analysis and rewrite performs the following tasks:

- Inserts calls to an accounting object before expensive bytecodes. The goal of this rewrite is to deterministically terminate code that has run for an unacceptably long amount of time or used an unacceptable amount of memory. Expensive bytecodes include method invocation, allocation, backwards jumps and throwing exceptions.
- Prevents exception handlers from catching `Throwable`, `Error` or `ThreadDeath`.
- Adjusts constant pool references to relink the code against a 'shadow' JDK, which duplicates a subset of the regular JDK but inside a dedicated sandbox package. The shadow JDK is missing functionality that contract code shouldn't have access to, such as file IO or external entropy.
- Sets the `strictfp` flag on all methods, which requires the JVM to do floating point arithmetic in a hardware independent fashion. Whilst we anticipate that floating point arithmetic is unlikely to feature in most smart contracts (big integer and big decimal libraries are available), it is available for those who want to use it.
- Forbids `invokedynamic` bytecode except in special cases, as the libraries that support this functionality have historically had security problems and it is primarily needed only by scripting languages. Support for the specific lambda and string concatenation metafactories used by Java code itself are allowed.
- Forbids native methods.
- Forbids finalizers.

The cost instrumentation strategy used is a simple one: just counting bytecodes that are known to be expensive to execute. Method size is limited and jumps count towards the budget, so such a strategy is guaranteed to eventually terminate. However it is still possible to construct bytecode sequences by hand that take excessive amounts of time to execute. The cost instrumentation is designed to ensure that infinite loops are terminated and that if the cost of verifying a transaction becomes unexpectedly large (e.g. contains algorithms with complexity exponential in transaction size) that all nodes agree precisely on when to quit. It is *not* intended as a protection against denial of service attacks. If a node is sending you transactions that appear designed to simply waste your CPU time then simply blocking that node is sufficient to solve the problem, given the lack of global broadcast.

Opcode budgets are separate per opcode type, so there is no unified cost model. Additionally the instrumentation is high overhead. A more sophisticated design would be to statically calculate bytecode costs as much as possible ahead of time,

by instrumenting only the entry point of 'accounting blocks', i.e. runs of basic blocks that end with either a method return or a backwards jump. Because only an abstract cost matters (this is not a profiler tool) and because the limits are expected to bet set relatively high, there is no need to instrument every basic block. Using the max of both sides of a branch is sufficient when neither branch target contains a backwards jump. This sort of design will be investigated if the per category opcode-at-a-time accounting turns out to be insufficient.

A further complexity comes from the need to constrain memory usage. The sandbox imposes a quota on bytes *allocated* rather than bytes *retained* in order to simplify the implementation. This strategy is unnecessarily harsh on smart contracts that churn large quantities of garbage yet have relatively small peak heap sizes and, again, it may be that in practice a more sophisticated strategy that integrates with the GC is required in order to set quotas to a usefully generic level.

Control over `Object.hashCode()` takes the form of new JNI calls that allow the JVM's thread local random number generator to be reseeded before execution begins. The seed is derived from the hash of the transaction being verified.

Finally, it is important to note that not just smart contract code is instrumented, but all code that it can transitively reach. In particular this means that the 'shadow JDK' is also instrumented and stored on disk ahead of time.

10 Notaries and consensus

Corda does not organise time into blocks. This is sometimes considered strange, given that it can be described as a blockchain system or 'blockchain inspired'. Instead a Corda network has one or more notary services which provide transaction ordering and timestamping services, thus abstracting the role miners play in other systems into a pluggable component.

Notaries are expected to be composed of multiple mutually distrusting parties who use a standard consensus algorithm. Notaries are identified by and sign with compound public keys (§5.1)that conceptually follow the Interledger Crypto-Conditions specification[7]. Note that whilst it would be conventional to use a BFT algorithm for a notary service, there is no requirement to do so and in cases where the legal system is sufficient to ensure protocol compliance a higher performance algorithm like RAFT may be used. Because multiple notaries can co-exist a single network may provide a single global BFT notary for general use and region-specific RAFT notaries for low latency trading within a unified regulatory area, for example London or New York.

10.1 Comparison to Nakamoto block chains

Bitcoin organises the timeline into a chain of blocks, with each block pointing to a previous block the miner has chosen to build upon. Blocks also contain a rough timestamp. Miners can choose to try and extend the block chain from any previous block, but are incentivised to build on the most recently announced block by the fact that other nodes in the system only recognise a block if it's a part of the chain with the most accumulated proof-of-work. As each block contains a reward of newly issued bitcoins, an unrecognised block represents a loss and a recognised block typically represents a profit.

Bitcoin uses proof-of-work because it has a design goal of allowing an unlimited number of identityless parties to join and leave the network at will, whilst simultaneously making it hard to execute sybil attacks (attacks in which one party creates multiple identities to gain undue influence over the network). This is an appropriate design to use for a peer to peer network formed of volunteers who can't/won't commit to any long term relationships up front, and in which identity verification is not done. Using proof-of-work then leads naturally to a requirement to quantise the timeline into chunks, due to the probabilistic nature of searching for a proof. The chunks must then be ordered relative to each other and the block chain algorithm follows as a result.

A Corda network is email-like in the sense that nodes have long term stable identities, which they can prove ownership of to others. Sybil attacks are blocked by the network entry process. This allows us to discard proof-of-work along with its multiple unfortunate downsides:

- Energy consumption is excessively high for such a simple task, being comparable at the time of writing to the consumption of an entire town. At a time when humanity needs to use less energy rather than more this is ecologically undesirable.
- High energy consumption forces concentration of mining power in regions with cheap or free electricity. This results in unpredictable geopolitical complexities that many users would rather do without.
- Identityless participants mean all transactions must be broadcast to all network nodes, as there's no reliable way to know who the miners are. This worsens privacy.
- The algorithm does not provide finality, only a probabilistic approximation, which is a poor fit for existing business and legal assumptions.
- It is theoretically possible for large numbers of miners or even all miners to drop out simultaneously without any protocol commitments being violated.

Once proof-of-work is disposed of there is no longer any need to quantise the timeline into blocks because conflicts can be resolved at the level of the individ-

ual transaction instead, and because the parties asserting the correctness of the ordering are known ahead of time regular signatures are sufficient.

10.2 Algorithmic agility

Consensus algorithms are a hot area of research and new algorithms are frequently developed that improve upon the state of the art. Unlike most distributed ledger systems Corda does not tightly integrate one specific approach. This is not only to support upgrades as new algorithms are developed, but also to reflect the fact that different tradeoffs may make sense for different situations and networks.

As a simple example, a notary that uses RAFT between nodes that are all within the same city will provide extremely good performance and latency, at the cost of being more exposed to malicious attacks or errors by whichever node has been elected leader. In situations where the members making up a distributed notary service are all large, regulated institutions that are not expected to try and corrupt the ledger in their own favour trading off security to gain performance may make sense. In other situations where existing legal or trust relationships are less robust, slower but byzantine fault tolerant algorithms like BFT-SMaRT[?] may be preferable. Alternatively hardware security features like Intel SGX®may be used to convert non-BFT algorithms into a more trusted form using remote attestation and hardware protection.

Being able to support multiple notaries in the same network has other advantages:

- It is possible to phase out notaries (i.e. sets of participants) that no longer wish to provide that service by migrating states.
- The scalability of the system can be increased by bringing online new notaries that run in parallel. As long as access to the ledger has some locality (i.e. states aren't constantly being migrated between notaries) this allows for the scalability limits of common consensus algorithms or node hardware to be worked around.
- In some but not all cases, regulatory constraints on data propagation can be respected by having jurisdictionally specific notaries. This would not work well when two jurisdictions have mutually incompatible constraints or for assets that may frequently travel around the world, but it can work when using the ledger to track the state of deals or other facts that are inherently region specific.
- Notaries can compete on their availability and performance.
- Users can pick between *validating* and *non-validating* notaries. See below.
- Separate networks can start independent and be merged together later.

10.3 Validating and non-validating notaries

Validating notaries resolve and fully check transactions they are asked to deconflict. Thus in the degenerate case of a network with just a single notary and without the use of any privacy features, they gain full visibility into every transaction. Non-validating notaries assume transaction validity and do not request transaction data or their dependencies beyond the list of states consumed. With such a notary it is possible for the ledger to become wedged, as anyone who knows the hash and index of a state may consume it without checks. If the cause of the problem is accidental, the incorrect data can be presented to a non-validating notary to convince it to roll back the commit, but if the error is malicious then states controlled by such a notary may become permanently corrupted.

It is therefore possible for users to select their preferred point on a privacy/security spectrum for each state individually depending on how they expect the data to be used. When the states are unlikely to live long or propagate far and the only entities who will learn their transaction hashes are somewhat trustworthy, the user may select to keep the data from the notary. For liquid assets a validating notary should always be used to prevent value destruction and theft if the transaction IDs leak.

10.4 Merging networks

Because there is no single block chain it becomes possible to merge two independent networks together by simply establishing two-way connectivity between their nodes then configuring each side to trust each others notaries and certificate authorities.

This ability may seem pointless: isn't the goal of a decentralised ledger to have a single global database for everyone? It is, but a practical route to reaching this end state is still required. It is often the case that organisations perceived by consumers as being a single company are in fact many different entities cross-licensing branding, striking deals with each other and doing internal trades with each other. This sort of setup can occur for regulatory reasons, tax reasons, due to a history of mergers or just through a sheer masochistic love of paperwork. Very large companies can therefore experience all the same synchronisation problems a decentralised ledger is intended to fix but purely within the bounds of the same organisation. In this situation the main problem to tackle is not malicious actors but rather heterogenous IT departments, varying software development practices, unlinked user directories and so on. Such organisations can benefit from gaining experience with the technology internally and cleaning up their own internal views of the world before tackling the larger problem of synchronising with the wider world as well.

When merging networks, both sides must trust that each other's notaries have

never signed double spends. When merging an organisation-private network into the global ledger it should be possible to simply rely on incentives to provide this guarantee: there is no point in a company double spending against itself. However, if more evidence is desired, a standalone notary could be run against a hardware security module with audit logging enabled. The notary itself would simply use a private database and run on a single machine, with the logs exported to the people running a global network for asynchronous post-hoc verification.

11 The vault

In any blockchain based system most nodes have a wallet, or as we call it, a vault.

The vault contains data extracted from the ledger that is considered *relevant* to the node's owner, stored in a form that can be easily queried and worked with. It also contains private key material that is needed to sign transactions consuming states in the vault. Like with a cryptocurrency wallet, the Corda vault understands how to create transactions that send value to someone else by combining asset states and possibly adding a change output that makes the values balance. This process is usually referred to as 'coin selection'. Coin selection can be a complex process. In Corda there are no per transaction network fees which is a significant source of complexity in other sysetms, however transactions must respect the fungibility rules in order to ensure that the issuer and reference data is preserved as the assets pass from hand to hand.

Advanced vault implementations may also perform splitting and merging of states in the background. The purpose of this is to increase the amount of transaction creation parallelism supported. Because signing a transaction may involve human intervention (see §13) and thus may take a significant amount of time, it can become important to be able to create multiple transactions in parallel. The vault must manage state 'soft locks' to prevent multiple transactions trying to use the same output simultaneously. Violation of a soft lock would result in a double spend being created and rejected by the notary. If a vault were to contain the entire cash balance of a user in just one state, there could only be a single transaction being constructed at once and this could impose unacceptable operational overheads on an organisation. By automatically creating send-to-self transactions that split the big state into multiple smaller states, the number of transactions that can be created in parallel is increased. Alternatively many tiny states may need to be consolidated into a smaller number of more valuable states in order to avoid hitting transaction size limits.

The vault is also responsible for managing scheduled events requested by node-relevant states when the implementing app has been installed (see §15).

11.1 Direct SQL access

A distributed ledger is ultimately just a shared database, albeit one with some fancy features. The following features are therefore highly desirable for improving the productivity of app developers:

- Ability to store private data linked to the semi-public data in the ledger.
- Ability to query the ledger data using widely understood tools like SQL.
- Ability to perform joins between entirely app-private data (like customer notes) and ledger data.
- Ability to define relational constraints and triggers on the underlying tables.
- Ability to do queries at particular points in time e.g. midnight last night.
- Re-use of industry standard and highly optimised database engines.
- Independence from any particular database engine, without walling off too many useful features.

Corda states are defined using a subset of the JVM bytecode language which includes annotations. The vault recognises annotations from the *Java Persistence Architecture* (JPA) specification defined in JSR 338[14]. These annotations define how a class maps to a relational table schema including which member is the primary key, what SQL types to map the fields to and so on. When a transaction is submitted to the vault by a flow, the vault finds states it considers relevant (i.e. which contains a key owned by the node) and the relevant CorDapp has been installed into the node as a plugin, the states are fed through an object relational mapper which generates SQL `UPDATE` and `INSERT` statements. Note that data is not deleted when states are consumed, however a join can be performed with a dedicated metadata table to eliminate consumed states from the dataset. This allows data to be queried at a point in time, with rows being evicted to historical tables using external tools.

Nodes come with an embedded database engine out of the box, but may also be configured to point to a separate RDBMS. The node stores not only state data but also all node working data in the database, including flow checkpoints. Thus the state of a node and all communications it is engaged in can be backed up by simply backing up the database itself. The JPA annotations are independent of any particular database engine or SQL dialect and thus states cannot use any proprietary column types or other features, however, because the the ORM is only used on the write paths users are free to connect to the backing database directly and issue SQL queries that utilise any features of their chosen database engine that they like. They can also create their own tables and create merged views of the underlying data for end user applications, as long as they don't impose any constraints that would prevent the node from syncing the database with the actual contents of the ledger.

States are arbitrary object graphs. Whilst nothing stops a state from containing multiple classes intended for different tables, it is typical that the relational representation will not be a direct translation of the object-graph representation. States are queried by the vault for the ORM mapped class to use, which will often skip ledger-specific data that's irrelevant to the user like opaque public keys and may expand single fields like an `Amount<Issued<Currency>>` type into multiple database columns.

It's worth noting here that although the vault only responds to JPA annotations it is often useful for states to be annotated in other ways, for instance to customise its mapping to XML/JSON, or to impose validation constraints[15]. These annotations won't affect the behaviour of the node directly but may be useful when working with states in surrounding software.

12 Clauses

TODO

13 Secure signing devices

13.1 Background

A common feature of digital financial systems and blockchain-type systems in particular is the use of secure client-side hardware to hold private keys and perform signing operations with them. Combined with a zero tolerance approach to transaction rollbacks, this is one of the ways they reduce overheads: by attempting to ensure that transaction authorisation is robust and secure, and thus that signatures are reliable.

Many banks have rolled out CAP (chip authentication program) readers which allow logins to online banking using a challenge/response protocol to a smartcard. The user is expected to type in the right codes and copy the responses back to the computer by hand. These devices are cheap, but tend to have small, unreliable, low resolution screens and can be subject to confusion attacks if there is malware on the PC, e.g. if the malware convinces the user they are performing a login challenge whereas in fact they are authorising a payment to a new account. The primary advantage is that the signing key is held in a robust and cheap smart card, so the device can be replaced without replacing the key.

The state-of-the-art in this space are devices like the TREZOR[16] by Satoshi Labs or the Ledger Blue. These were developed by and for the Bitcoin community. They are more expensive than CAP readers and feature better screens and USB connections to eliminate typing. Advanced devices like the Ledger Blue

support NFC and Bluetooth as well. These devices differ from CAP readers in another key respect: instead of signing arbitrary, small challenge numbers, they actually understand the native transaction format of the Bitcoin network and parse the transaction to figure out the message to present to the user, who then confirms that they wish to perform the action printed on the screen by simply pressing a button. The transaction is then signed internally before being passed back to the PC via the USB/NFC/Bluetooth connection.

This setup means that rather than having a small device that authorises to a powerful server (which controls all your assets), the device itself controls the assets. As there is no smartcard equivalent the private key can be exported off the device by writing it down in the form of "wallet words": 12 random words derived from the contents of the key. Because elliptic curve private keys are small (256 bits), this is not as tedious as it would be with the much larger RSA keys the financial industry is typically using.

There are clear benefits to having signing keys be kept on personal, employee-controlled devices only, with the organisation's node not having any ability to sign for transactions itself:

- If the node is hacked by a malicious intruder or bad insider they cannot steal assets, modify agreements, or do anything else that requires human approval, because they don't have access to the signing keys. There is no single point of failure from a key management perspective.
- It's more clear who signed off on a particular action - the signatures prove which devices were used to sign off on an action. There can't be any back doors or administrator tools which can create transactions on behalf of someone else.
- Devices that integrate fingerprint readers and other biometric authentication could further increase trust by making it harder for employees to share/swap devices. In theory, an iPhone or Android device could be used as a transaction authenticator, although this would be expensive.

13.2 Confusion attacks

The biggest problem facing anyone wanting to integrate smart signing devices into a distributed ledger system is how the device processes transactions. For Bitcoin it's straightforward for devices to process transactions directly because their format is very small and simple (in theory - in practice a fixable quirk of the Bitcoin protocol actually significantly complicates how these devices must work). Thus turning a Bitcoin transaction into a human meaningful confirmation screen is quite easy:

```
Confirm payment of 1.23 BTC to 1AbCd0123456.......
```

This confirmation message is susceptible to confusion attacks because the opaque

payment address is unpredictable. A sufficiently smart virus/attacker could have swapped out a legitimate address of a legitimate counterparty you are expecting to pay with one of their own, thus you'd pay the right amount to the wrong place. The same problem can affect financial authenticators that verify IBANs and other account numbers: the user's source of the IBAN may be an email or website they are viewing through the compromised machine. The BIP 70[10] protocol was designed to address this attack by allowing a certificate chain to be presented that linked a target key with a stable, human meaningful and verified identity.

For a generic ledger we are faced with the additional problem that transactions may be of many different types, including new types created after the device was manufactured. Thus creating a succinct confirmation message inside the device would become an ever-changing problem requiring frequent firmware updates. As firmware upgrades are a potential weak point in any secure hardware scheme, it would be ideal to minimise their number.

13.3 Transaction summaries

To solve this problem we add a top level summaries field to the transaction format (joining inputs, outputs, commands, attachments etc). This new top level field is a list of strings. Smart contracts get a new responsibility. They are expected to generate an English message describing what the transaction is doing, and then check that it is present in the transaction. The field is a list of strings rather than a single string because a transaction may do multiple things simultaneously in advanced use cases.

Because the calculation of the confirmation message has now been moved to the smart contract itself, and is a part of the transaction, the transaction can be sent to the signing device: all it needs to do is extract the messages and print them to the screen with YES/NO buttons available to decide whether to sign or not. Because the device's signature covers the messages, and the messages are checked by the contract based on the machine readable data in the states, we can know that the message was correct and legitimate.

The design above is simple but has the issue that large amounts of data are sent to the device which it doesn't need. As it's common for signing devices to have constrained memory, it would be unfortunate if the complexity of a transaction ended up being limited by the RAM available in the users signing devices. To solve this we can use the tear-offs mechanism (see §5.6) to present only the summaries and the Merkle branch connecting them to the root. The device can then sign the entire transaction contents having seen only the textual summaries, knowing that the states will trigger the contracts which will trigger the summary checks, thus the signature covers the machine-understandable version of the transaction as well.

Note, we assume here that contracts are not themselves malicious. Whilst a

malicious user could construct a contract that generated misleading messages, for a user to see states in their vault and work with them requires the accompanying CorDapp to be loaded into the node as a plugin and thus whitelisted. There is never a case where the user may be asked to sign a transaction involving contracts they have not previously approved, even though the node may execute such contracts as part of verifying transaction dependencies.

13.4 Identity substitution

Contract code only works with opaque representations of public keys. Because transactions in a chain of custody may need to be anonymised, it isn't possible for a contract to access identity information from inside the sandbox. Therefore it cannot generate a complete message that includes human meaningful identity names even if the node itself does have this information.

To solve this the messages placed inside a transaction may contain numeric indexes of the public keys required by the commands using backtick syntax. The transaction is provided to the device along with the X.509 certificate chains linking the pseudonymous public keys to the long term identity certificates, which for transactions involving the user should always be available (as they by definition know who their trading counterparties are). The device can verify those certificate chains to build up a mapping of index to human readable name, and then perform the message substitution before rendering. Care must be taken to ensure that the X.500 names issued to network participants do not contain text chosen to deliberately confuse users, e.g. names that contain quote marks, partial instructions, special symbols and so on. This can be enforced at the network permissioning level.

13.5 Multi-lingual support

The contract is expected to generate a human readable version of the transaction. This should be in English, by convention. In theory, we could define the transaction format to support messages in different languages, and if the contract supported that the right language could then be picked by the signing device. However, care must be taken to ensure that the message the user sees in alternative languages is correctly translated and not subject to ambiguity or confusion, as otherwise exploitable confusion attacks may arise.

14 Client RPC and reactive collections

TODO

15 Event scheduling

TODO

16 Future work

TODO

Secure hardware

Zero knowledge proofs

17 Conclusion

TODO

18 Acknowledgements

TODO

Bibliography

[1] Brown, Carlyle, Grigg, Hearn. *Corda: An introduction.* `http://r3cev.com/s/corda-introductory-whitepaper-final.pdf`, 2016.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[3] Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* `https://bitcoin.org/bitcoin.pdf`, 2008.

[4] Lindholm, Yellin, Bracha, & Buckley. *The Java Virtual Machine Specification Java SE 8 Edition.* `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`, 2015.

[5] Buterin et al. *A Next-Generation Smart Contract and Decentralized Application Platform.* https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper, 2014.

[6] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. Cryptology ePrint Archive, Report 2016/199, 2016. http://eprint.iacr.org/2016/199.

[7] Stefan Thomas. Crypto-conditions. https://interledger.org/five-bells-condition/spec.html, 2016.

[8] OASIS. Advanced message queuing protocol (amqp) version 1.0, 2012.

[9] Mike Hearn. Bitcoin micropayment channels. https://bitcoinj.github.io/working-with-micropayments, 2014.

[10] Mike Hearn, Gavin Andresen. Bitcoin payment protocol. https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki, 2013.

[11] java.time.Instant documentation. https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html, 2014.

[12] PKWARE. Zip file format. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT, 1989.

[13] Pieter Wiulle. Hierarchical deterministic wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2013.

[14] Jsr 338: Java persistence api. http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf?AuthParam=1478095024_77b7362fd5bd185ebf8d2cd2a071a14d, 2013.

[15] Jsr 349: Bean validation constraints. https://www.jcp.org/en/jsr/detail?id=349, 2013.

[16] Bitcoin trezor device. https://bitcointrezor.com/, 2016.