Polyglot Canton
Towards an open language ecosystem and EVM compatibility with privacy on Canton Network

Digital Asset
info@digitalassset.com
https://digitalasset.com
https://canton.network

February 2025

Abstract

Canton Network is a public blockchain network with configurable controls and privacy. Thanks to a unique stakeholder-based consensus methodology, it can be set up to serve anything from public decentralized applications, to addressing the requirements demanded by regulators and financial institutions. It can do all this while still maintaining blockchain's key ability to transact atomically and with low trust across the entire public network. The consensus protocol builds on an extended UTXO ledger model in which transactions have a rich structure that allows their decomposition into overlapping sub-transactions that can be distributed and validated independently in a deterministic fashion. A rich ledger model such as this puts strong demands on the smart contract language that is used both to construct and validate the resulting transactions. The solution in Canton to date is the Daml smart contract language, a custom language stack derived from Haskell. The Daml language has allowed Canton to be proven out and mature, but for a public and open Canton Network, there are good reasons to open it up to alternative programming experiences. Firstly, Daml is a purely functional and strongly statically typed language designed to provide maximal confidence and safety for mission critical institutional use cases. Supporting additional languages following other proven paradigms would make Canton accessible to a wider pool of developers requiring less upfront education. Secondly, Canton Network aims to apply the best parts of crypto and DeFi to traditional finance, and to break down the currently hard barrier between those two worlds. Supporting those languages that have become popular in DeFi, Solidity in particular, would further that aim, and additionally make those languages compatible with the control and privacy requirements demanded in regulated finance. And lastly, and maybe most crucially, smart contract and general language ecosystems have evolved over recent years to the point where there are compelling and viable alternatives to Daml's current language stack. This paper demonstrates alternative language engines like Wasm in Canton, which in turn opens the door to compilation or hosting of Solidity in Wasm using proven tools like Solang or Rust-EVM. It presents a path to a future where Canton is polyglot, widely accessible, and compatible with EVM chains and DeFi.

1 Introduction

Blockchains and smart contract based applications promise to transform financial technology[1,2]. Today's financial systems operate in one of two ways. The model most prevalent in consumer facing digital native finance is centralization of data and control in large financial intermediaries that can offer slick near real time experiences. Services like PayPal or Robinhood are examples of such services, including the downsides of the added counterparty risks. The second model, prevalent in traditional finance, predates even the internet. Every financial institution keeps its own books. Ownership and servicing of assets is managed through complex hierarchical account structures across multiple organizations. Transactions that operate on multiple organizations' books have to be coordinated through long sequences of messages between these organizations, and errors corrected through reconciliation processes. This system is distributed, and for all intents and purposes even trustless and decentralized, but consistency across organizations is ascertained through human error checking and labor, not through properties of the system itself.

Blockchain, first introduced by Bitcoin[3], offers a novel third way. Ownership records are kept on a low trust distributed system, often called a *decentralized* system in a way that combines the benefits of the two existing approaches. The system operates in near real time allowing investor experiences similar to those on centralized systems. Yet the system is so secure that participants can treat it with the same confidence as a book they would keep themselves and thus use it as a shared golden source of truth for the state of assets, all without centralization of book keeping or trust in any centralized entity. This is so robust and proven that in the United States of America, Bitcoin is considered a commodity[4].

In the years immediately following Bitcoin's introduction, there were a number of innovations and imitations all of which shared the key property that blockchain and application were one and the same. The Bitcoin application is part of the Bitcoin blockchain and the Bitcoin blockchain is not optimized to host any other application. This paradigm was shifted by Ethereum[5], widely regarded as the first general smart contract blockchain. Buterin, the author of the Ethereum paper, has stated that the name "smart contract" is a bad choice, proposing instead "persistent script" as a name[6]. This latter name does capture the idea rather well. Smart contracts allow developers to express state schemas for persisted data that is synchronized via the blockchain, as well as scripts which operate on that data, corresponding to rules and authorizations of those rules which govern the data. Persisted data, and procedures on data are the essence of an application backend and as such Ethereum was highly successful in enabling blockchain applications: third party applications developed with smart contracts as the core persistence and business logic.

[1] III. Blueprint for the future monetary system: improving the old, enabling the new, BIS, 2023, https://www.bis.org/publ/arpdf/ar2023e3.pdf
[2] Finternet: the financial system for the future, Carstens & Nilekani, 2024, https://www.bis.org/publ/work1178.pdf
[3] Bitcoin: A Peer-to-Peer Electronic Cash System, Sathoshi Nakamoto, 2009, https://bitcoin.org/bitcoin.pdf
[4] BitCoin basics, CFTC, 2019, https://www.cftc.gov/sites/default/files/2019-12/oceo_bitcoinbasics0218.pdf
[5] Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, Buterin, 2014, https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf
[6] Twitter (now X) post, Buterin, 2018, https://x.com/vitalikbuterin/status/1051160932699770882?lang=en

While already of high value for single applications, smart contract chains like Ethereum typically unlock further benefits by acting as an application platform for multiple applications. Applications hosted on the same blockchain can interact with each other by making programmatic calls from one smart contract to another, with the guarantee that any effects across smart contracts are committed in a single atomic transaction. In terms of developer experience, this is as simple as calling one C library from another, and collating effects in a single database transaction, but thanks to the decentralized nature of the system, the shared persistence, and low trust, it opens up entirely new possibilities. One developer can launch a payment system. Another developer can launch a tokenized bond. And a third can launch an exchange, in which these assets can be swapped, instantly, atomically, and without counterparty risks. This is commonly referred to as a delivery versus payment transaction, or DvP. All of these applications can be run with centralized or decentralized control, giving rise to entirely new and innovative business models, commonly referred to as Decentralized Finance (DeFi).

To illustrate the difference between traditional app integration via APIs versus composition on the blockchain, consider travel aggregators. Alice wants to book a complex holiday and searches for flights on several airlines, hotels, and car rental through an aggregator. The prices are shown and she decides to book. After entering her payment details, the aggregator in the background attempts to make independent bookings with the multiple airlines, hotels, and car rental companies. It's possible for some of these to succeed, and some of these to fail, leaving Alice with a partial itinerary. She now has to plug the gap, potentially leaving her with less favorable terms than an altogether different itinerary. Contrast what an experience on a blockchain system could look like. The settlement, meaning the exchange of Alice's money for binding bookings with all the different vendors, can happen in a single atomic transaction. If one booking fails, all fail, leaving Alice free to pursue a different arrangement.

Blockchain for regulated finance can solve the same two problems that the technology demonstrably solves in crypto and DeFi: Synchronization in real time with low trust and high integrity, and unparalleled interoperability between applications running on the same network. But there are additional challenges for adoption of the technology by regulated entities, and these have been understood in some form for as long as smart contract blockchains have been around.

Regulated institutions are regulated precisely to enforce some baseline of risk management that ensures financial stability. Distributed systems across multiple institutions introduce new counterparties, or new ways of interacting with those counterparties, and that can significantly change the risk profile of assets managed on such systems. The regulators have clarified the treatment of blockchain based assets over the last years[7], and have identified the specific risks associated with the technology as used in crypto and DeFi, commonly called *public permissionless*[8] networks. Taking the above example of a payment and tokenized bond being exchanged in a DvP transaction, for the assets in question to be considered equivalent to the same assets managed on

[7] SCO60 Scope and Definitions on Cryptoasset Exposures, BIS, 2022, https://www.bis.org/basel_framework/chapter/SCO/60.htm
[8] Working paper 44 Novel risks, mitigants and uncertainties with permissionless distributed ledger technologies, BIS, 2024, https://www.bis.org/bcbs/publ/wp44.pdf

traditional IT systems, they need equivalent risk profiles. Each asset needs to be backed by a clearly accountable and licensed registrar[9]. That registrar needs sufficient *control* to ensure settlement, finality and legal compliance. And the registrar needs to maintain *privacy and confidentiality*, preventing ownership data leaking between applications and between investors.

These requirements gave rise to a first generation of enterprise smart contract blockchain systems soon after the launch of Ethereum, most notably Quorum[10], Besu[11], Fabric[12], and Corda[13]. First and foremost, these systems solve for the regulatory control requirements by running as *private permissioned* networks, and this has been proven out numerous times with regulated production systems running on all of the above systems around the world[14]. However, confidentiality (often referred to as privacy) remains a challenge for many such systems[15], and due to the lack of interoperability between private systems, the network effects visible in DeFi are not materializing in the same way. A number of initiatives are attempting to address this issue[16] and establish *unified ledgers* that can act as the common venue for multiple regulated assets and services.

Privacy and confidentiality also remain one of the greatest challenges for the Ethereum ecosystem[17] and public permissioned networks in general. An example where this is creating challenges in practice is that collateral movements in and out of crypto derivatives exchanges are fully transparent on chain. That means anyone can use this information plainly at regular intervals to deduce a trader or market maker's financial position, and use that information to their advantage.
The blockchain community at large is pursuing a number of approaches to add privacy and confidentiality to the Ethereum ecosystem. The below lists a few of the prevalent ideas with their capabilities and limitations. They all have in common that they try to add privacy *on top of* the Ethereum Virtual Machine (EVM) and as such do not address the core architecture of EVM chains, which is a fully replicated, transparent, and permissionless consensus. As a consequence, none of them are able to meet general purpose privacy and confidentiality on Solidity contracts:

- Anonymization, pseudonymization and similar approaches like the Stealth Addresses cited in (An incomplete guide to stealth addresses, Buterin, 2023)
- Privacy Pools[18], which use zero knowledge proof cryptography for simple tokenization by hiding the connections between deposits and withdrawals from a pool of assets.
- Homomorphic encryption[19] offers the obfuscation of integer values while still operating on them using basic arithmetic operations and comparisons.

[9] Registrar on Investopedia, https://www.investopedia.com/terms/r/registrar.asp
[10] JP Morgan's Quorum blockchain powers new correspondent banking network, www.bankingtech.com, 2017, https://web.archive.org/web/20171109080854/http://www.bankingtech.com/1037032/jp-morgans-quorum-blockchain-powers-new-correspondent-banking-network/
[11] Hyperledger Unanimously Approves First Ethereum Codebase For Enterprises, Forbes, 2019, https://www.forbes.com/sites/michaeldelcastillo/2019/08/29/hyperledger-unanimously-approves-first-ethereum-codebase-for-enterprises/
[12] Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains, Elli Androulaki et al, 2018, https://arxiv.org/abs/1801.10228
[13] Corda: An introduction, Brown / Carlyle / Grigg/ Hearn, 2016, https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf
[14] Corda Use Case Directory, R3, https://r3.com/products/use-case-directory-all/
[15] Blockchain privacy delays launch of Brazil's DREX CBDC, enters phase 2, Ledger Insights, 2024, https://www.ledgerinsights.com/blockchain-privacy-delays-launch-of-brazils-drex-cbdc-enters-phase-2/
[16] Global Layer 1 (GL1) Whitepaper, Monetary Authority of Singapore, 2024, https://www.mas.gov.sg/publications/monographs-or-information-paper/2024/gl1-whitepaper
[17] An incomplete guide to stealth addresses, Buterin, 2023, https://vitalik.eth.limo/general/2023/01/20/stealth.html
[18] Blockchain Privacy and Regulatory Compliance: Towards a Practical Equilibrium, Buterin et. al, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4563364
[19] fhEVM github repo, https://github.com/zama-ai/fhevm

- Private chains or rollups, which restrict access to the blockchain to a small invite-only group of participants. This provides privacy from the general public, but there is no confidentiality within the chain. It's fully transparent for anyone with access. Traditionally, this was the domain of enterprise ethereum clients like Quorum and Besu, but Tessera, one of the last remaining private transaction managers for Ethereum, was recently sunset[20]. Private rollups, subnets, and sidechains offer the same level of privacy, but slightly better interoperability in that they typically have inbuilt non-transactional messaging to other chains in the same ecosystem.
- Confidential compute[21] can be used to run the blockchain in hardware enclaves provided by groups of semi-trusted entities. In this model, data is either public, meaning visible to all, or private, visible to nobody at all.

Notably and slightly surprisingly given the privacy rhetoric around zero knowledge proofs, the authors of this paper could *not* find any active projects attempting to add confidentiality or privacy to regular Ethereum contracts.

Canton[22] is a next generation layer 1 with fine-grained smart-contract configured controls, and need-to-know privacy and confidentiality based on data minimization, allowing everything from permissionless DeFi to regulated finance, all while maintaining the ability to perform atomic smart contract calls between independent applications. It enables application composition along the lines of the DeFi example above for regulated entities. Two regulated institutions can launch payment and bond tokenization applications maintaining full privacy, confidentiality and control. Asset owners can construct and execute atomic transactions which move assets on both sides without affecting the privacy or confidentiality properties. It allows the organic development of a public network and a unified ledger where applications and users can openly join and extend the business running on the network like in a public permissionless chain, but where individual applications can be controlled in the same vein as on a private permissioned network. Picking up on the collateral example above, a transaction pledging an asset to an exchange would be precisely to the trader, the exchange, and escrow agent at which the asset is pledged, and the institution(s) running the asset's registry. Returning to the DvP example, Canton can execute a DvP with *sub-transaction privacy*. The buyer and owner see the entire swap, but the registrars of payment and delivery assets see only a simple transfer of their respective assets, all while still guaranteeing atomic settlement, and providing resilience against malicious participants.

Canton's ledger model (formerly the Daml Ledger Model[23]) provides both the abstract conceptual mental model for smart contract developers on Canton, as well as the theoretical foundation for Canton to offer its independent control, privacy, and confidentiality properties. Like many other privacy-first blockchain platforms (see section 2.3), Canton uses an extended unspent transaction output model (eUTXO). The UTXO

[20] Sunsetting Tessera and Simplifying Besu, Linux Foundation, 2024, https://www.lfdecentralizedtrust.org/blog/sunsetting-tessera-and-simplifying-hyperledger-besu
[21] Oasis Sapphire network, https://oasisprotocol.org/sapphire
[22] Canton: A Daml based ledger interoperability protocol, Digital Asset, 2020, https://www.canton.io/publications/canton-whitepaper.pdf
[23] Daml documentation on the Daml Ledger Model, https://docs.daml.com/concepts/ledger-model/index.html

model goes back all the way to Bitcoin and is based on the idea that the ledger state is simply the set of immutable outputs from committed transactions that have not yet been *spent* by later transactions. Canton extends this model by enriching transactions with a hierarchical structure of *actions*, which can be thought of as the call graph of smart contract functions, including special functions for the creation and spending of outputs. As part of the consensus protocol, this call graph is decomposed into stakeholder specific views, which are distributed on a need to know basis using standard cryptography and data minimization, and validated using a fine grained stakeholder based Proof of Authority per view. This decomposition of transactions translates into a decomposition of ledger state so that each participant holds only *their* subset of all UTXOs on a need to know basis, thus providing privacy and confidentiality.

Abstract ledger models are common to many smart contract ledgers, and the EVM ledger model provides a good point of comparison. In the EVM ledger model, the state consists of Accounts (identified by addresses), which can be either externally owned via a cryptographic key, or hold smart contract code and mutable state controlled by that code. All accounts are visible to all participants in a fully transparent fashion, which gives each smart contract account, and indeed the full EVM, the properties of a fully replicated state machine. A transaction is a top level function call into a smart contract account, which every participant can execute consistently due to their knowledge of the entire global state, moving the EVM state machine from one state to the next.

The addition of the primitives for a ledger model is what turns a surface language into a smart contract language. For example, Solidity extends an ECMAScript expression language with primitives for EVM smart contract account. The Solana Program Crate[24] adds Solana ledger model primitives to Rust to turn Rust into a smart contract language. The more cleanly the ledger model is separated from the language interpreter, the easier it is to support additional languages and virtual machines. As Section 4 will demonstrate, the Canton ledger model is well enough abstracted out to allow for additional languages and virtual machines. The EVM ledger model, by contrast, is relatively tightly intertwined with the EVM interpreter, which is why efforts for additional languages on EVM ledgers are typically approached by compiling to EVM (e.g. Vyper[25]), not by running additional VMs side-by-side with the EVM.

The Daml smart contract language[26] was developed in parallel with Canton from the outset to expose the primitives of Canton's ledger model. It is purpose made to abstract away the complexities of the protocol and allow developers to concisely express the shared data, rules, and permissions of multiparty workflows, including financial applications. Daml has been proven in some of the highest volume Distributed Ledger Technology (DLT) applications in the world[27]. Daml is, and will remain, a robust choice for programming enterprise grade smart contract applications on Canton.

Daml's tech stack and design choices were driven by a combination of factors, including a desire to provide a high degree of developer safety, requirements on runtime safety and determinism, and the available language and compiler tools at the time. Some

[24] Solana Program Crate, https://crates.io/crates/solana-program
[25] Vyper documentation, https://docs.vyperlang.org/en/stable/
[26] Daml: A Smart Contract Language for Securely Automating Real-World Multi-Party Business Workflows, Bernauer et al, 2019, https://arxiv.org/abs/2303.03749
[27] DLR Transacts $1 Trillion a Month, Broadridge, 2023, https://www.broadridge.com/article/capital-markets/dlr-transacts-1-trillion-a-month

of these design drivers, like runtime safety and determinism, hold up. Others, like strong static typing and a purely functional expression language are a matter of taste and can present a hurdle to developers not familiar with those language paradigms[28,29]. And for some, new alternatives are available, in that robust, safe, and deterministic runtime environments like WebAssembly (Wasm) engines have been developed and proven out in the meantime. If Canton were developed today, it would likely take advantage of Wasm from the outset, and trade off some safety for accessibility in its surface language.

Opening up Canton to additional smart contract languages is feasible thanks to the clean abstract ledger model, and these advancements in the language space. Additional development experiences will ease access to Canton for new, larger pools of developers, and make Canton's privacy and control properties available to those developers.

The availability of near-deterministic byte codes and virtual machines like Wasm, as well as low-level or purpose-made languages like Rust and AssemblyScript that do not rely on any particular Wasm host functions, provide a solid language and runtime foundation. This enables programming models for Canton akin to the Solana Program Crate referenced above, exposing ledger model primitives through libraries in standard languages like Rust. Rust's ecosystem of developer tools, its familiar curly brace syntax, and imperative style is likely more accessible and attractive to developers without a background in pure familiar programming.

Developer accessibility is also important for the existing community of smart contract developers. The success of DeFi on Ethereum has resulted in its smart contract language, Solidity, becoming quite dominant. Solidity is as intertwined with Ethereum's ledger model as Daml is with Canton's. As such, it doesn't make sense to consider Solidity as the *primary* programming language for Canton. However, supporting Solidity as *one option* on Canton would have a number of major benefits. Solidity developers could adopt Canton with direct skills transfer, transitioning to native smart contracts only as the need arises. Applications already built for Ethereum, private permissioned deployments, in particular, could be lifted and shifted across to Canton Network and benefit from network effects that are possible there. And Canton could give Solidity contracts the properties that they lack for regulated use on other public networks: controls, privacy, and confidentiality while maintaining atomic smart contract calls.

The gravity of the privacy problem for Ethereum Virtual Machine (EVM) ledgers, and the partiality of all of the above approaches presents a great opportunity for the Canton Network. Canton supports fine grained sub-transaction privacy with smart contract interoperability at the ledger model and protocol level. EVM compatibility for Canton could give rise to a new type of privacy solution for EVM, in which different contracts or even pieces of contract state are accessible to different stakeholder groups, while maintaining atomic smart contract calls.

Since Daml, Wasm-native languages, and existing smart contract languages like Solidity offer varied benefits and styles attractive to different stakeholder groups, a truly open Canton requires *multiple* and *additional* languages. Daml needs to be decoupled

[28] Learning Daml: Advantages and Challenges, Behnke, 2023, https://www.halborn.com/blog/post/learning-daml-advantages-and-challenges
[29] Empirical Analysis of Programming Language Adoption, Meyerovich and Rabkin, 2013, https://lmeyerov.github.io/projects/socioplt/papers/oopsla2013.pdf

from Canton even further to turn Canton into a truly polyglot ecosystem where different developers can choose their smart contract language to suit their needs, skills, and use cases.

This paper is laid out in four sections diving deeply into the status quo, ongoing work, and future opportunities regarding the languages supported by Canton.

Section 2 will recap those parts of the Canton protocol required to understand smart contract language design and considerations. Section 2.1 presents the high level architecture of Canton, in particular its two-tier network design consisting of participant nodes and synchronizers. These correspond roughly to a separation of validation and ordering, or to full nodes and low level networking and byzantine fault tolerance (BFT) networking in other networks. Section 2.2 covers identity and cryptography as needed to understand the abstract identity concept of *Parties* introduced in the Canton ledger model in section 2.3. The latter is an abstract extended unspent transaction output (eUTXO) model, which not only extends what data and script a UTXO can hold, but also endows transactions with significantly more structure than usual eUTXO models, tracking smart contract calls in a tree of *actions*. This structure is used in Canton's transaction protocol, covered in section 2.4, to decompose transactions into *views* that can be distributed and validated independently of each other. This decomposition allows independent applications to exert control while maintaining confidentiality through *sub-transaction privacy*. Section 2.5 concludes the discussion on Canton by covering how smart contracts fit into the consensus and connect it with the abstract ledger model. Smart contract *packages* are introduced as the deployable unit of smart contract code, and key constraints and requirements on the language stack are covered in preparation for the rest of the paper.

Section 3 presents the Daml smart contract language. Section 3.1 shows an example of a typical Daml package, how it maps to the ledger model, and how a smart contract call translates into a transaction. The simple example chosen acts as a baseline for comparison with future languages, as well as to provide some familiarity to readers to read Daml code snippets in later chapters. Section 3.2 introduces the Daml language stack at a level needed to consider additional languages being supported by presenting the key components and interfaces with Canton that need to be modified, extended, or opened up to support additional languages, most notably Daml's intermediary language Daml-LF, its interpreter, and how it interacts with Canton.

Sections 4 and 5 present two parallel but interrelated bodies of work to open up Canton to additional languages. Section 4 discusses how Canton's intermediary language and Canton's interaction with smart contract engines (commonly called Virtual Machines, or VMs) could be modified to support additional VMs. Specifically, section 4.1 presents work on Canton smart contracts written in Rust, and compiled to and executed in a Wasm runtime. Section 4.2 presents how this capability could be used to support an accessible high level language experience in Rust, or other Wasm-native languages like AssemblyScript to develop Canton smart contracts, using standard IDE and compiler tooling. The construction of 4.1 and 4.2 is such that it generalizes to many runtimes and

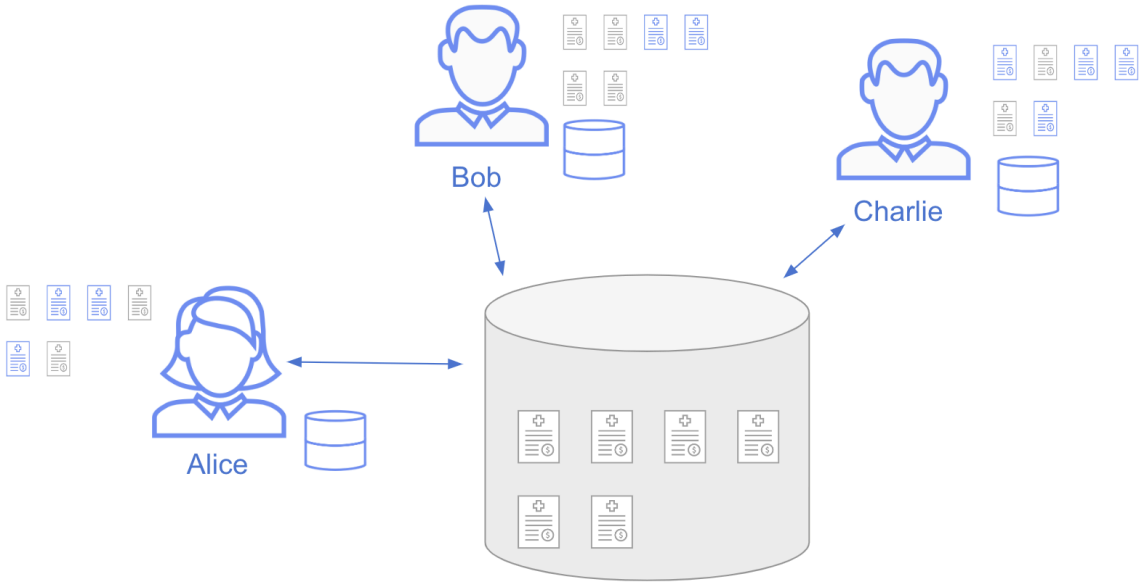
languages and thus presents a path towards Canton supporting numerous VMs and language stacks, possibly even in a pluggable fashion.

Section 5 covers Solidity specifically. Solidity is about as intertwined with Ethereum and the Ethereum Virtual Machine (EVM) as Daml is with Canton. Unlike support of a language like Rust or AssemblyScript, supporting Solidity isn't just about language support, but about ledger model and API support. Section 5.1 uses an example of porting SocGen's ForgeBond contract[30] on Ethereum to Canton and discusses how pure EVM support may be feasible based on section 4's Wasm work, by either cross-compiling Solidity to Wasm using the open source Hyperledger Solang[31] compiler, or by hosting a Wasm-based EVM like Rust-EVM[32]. Section 5.2 goes on to discuss how the EVM ledger model could be mapped to the Canton ledger model to go from surface language support to smart contract support. The section discusses challenges like contention, as well as advantages over standard EVM, like the ability to run atomic transactions between two private Solidity contracts. Section 5.3 discusses API compatibility to achieve functional lift and shift of Solidity applications to Canton. Finally, section 5.4 envisions how minor language extensions to Solidity could be used to inform a compiler or EVM how to map Solidity contracts to Canton's ledger model in more sophisticated ways, which may allow not only scalability close to Canton-native smart contracts, but for Solidity contracts to benefit from Canton's full sub-transaction privacy.

2 Canton

Canton is a blockchain or Distributed Ledger Technology (DLT) system which gives multiple parties consistent views into a *virtual* global ledger. The global ledger is not materialized in any one place, but kept consistent through stakeholder based consensus between the participants according to rules laid out in smart contracts. Participants' views are materialized locally on their own nodes, and used as a permanent record of data. The below image shows participants Alice, Bob, and Charlie at a conceptual level connecting to a virtual global ledger and holding different views of that ledger.

[30] ForgeBond source code via Blockscan, https://vscode.blockscan.com/ethereum/0x1Ff3D45E2c6c638A8d6BD1c81c99E6dB6D585EEb
[31] Hyperledger Solang, GitHub repo: https://github.com/hyperledger/solang, Docs: https://solang.readthedocs.io/
[32] Rust Evm GitHub repo: https://github.com/rust-ethereum/evm



This section offers a recap of Canton's architecture, cryptography and identity, ledger model, and consensus protocol as a foundation for the later discussions on smart contracts within Canton. Covering Canton's design and function to the level required to understand all the nuances involved in integrating smart contract languages goes beyond the scope of this paper. Readers that would like to dig deeper should refer to previous works[33,34] and documentation[35].

2.1 Architecture

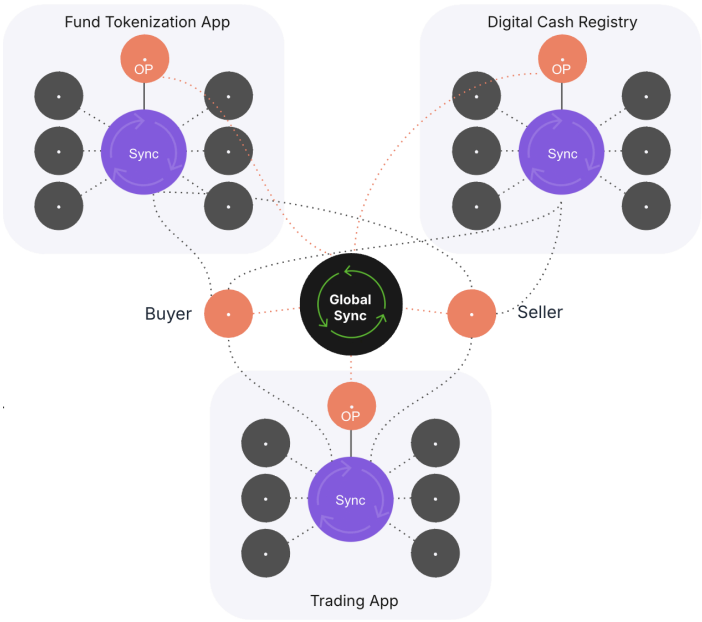
Nodes in the Canton Network are called *participant nodes*. A user or company deploys one or more participant nodes. The user's participant node stores the user's private transaction and state data, submits transactions to the network for the user, and participates in consensus on transactions in which the user is entitled or required to take part. In short, the participant node represents an independent participant at the protocol level. What most would consider "the ledger", the graph of transactions, resulting state, and cryptographic evidence, is all stored and processed amongst the participants. In both of these respects, a Canton participant is akin to a full node in traditional blockchains like Bitcoin or Ethereum. What sets Canton apart from most other blockchains is that both data and consensus are *segmented*, not fully replicated. Each node only holds its *view* of the ledger, so that no node holds the entire ledger. And consensus is run on a transaction by transaction basis, involving exactly the *stakeholder* participants in each transaction in a *per-transaction* Proof-of-Authority model.

To transport data between nodes and determine the order of messages, each participant node connects to one or more private or public *synchronizers* (previously

[33] Canton: A Daml based ledger interoperability protocol, Digital Asset, 2020, https://www.canton.io/publications/canton-whitepaper.pdf
[34] Canton Network: A Network of Networks for Smart Contract Applications, Digital Asset, 2024, https://www.digitalasset.com/hubfs/Canton/Canton%20Network%20-%20White%20Paper.pdf
[35] Daml documentation, Digital Asset, https://docs.daml.com/

synchronization domains, or sync domains). Any group of users can coordinate an atomic transaction amongst the group as long as their participant nodes are connected to a common synchronizer. Anyone can deploy synchronizers at will. To promote privacy and net neutrality, data in transit over sync domains is encrypted so that it can be decrypted in a strict need-to-know fashion, preventing operators or uninvolved participants from accessing message contents. Synchronizers can be thought of as highly available, fault-tolerant messaging queues between participant nodes that sequence, timestamp, and distribute encrypted messages to participant nodes with high transparency and ordering guarantees.

Transactions coordinated on one synchronizer can use the outputs from transactions coordinated on another synchronizer as inputs as long as the stakeholder participants in those outputs are connected to both synchronizers. This allows the transaction graph to fluidly span across synchronizers. This gives Canton a network of networks topology at the network level, while exposing a view into a single global ledger of transactions through the participants at the data level. Application providers and users can freely choose which synchronizers they use to coordinate which transactions. As such, Canton creates a mesh network of composable Daml applications in which each application may make different tradeoffs between trust, performance, access control, and operational complexity.



The resulting network is *public* by providing open, internet-like extensibility of this mesh of participants, synchronizers and applications as well as through the existence of open synchronizers and applications. But it is also *permissioned* as each participant, application, and synchronizer has a lot of independent control over which parts of the network they allow to interact with them and how, allowing a range of configurations from private centralized subnetworks and services with strong controls to public and decentralized network infrastructure and DeFi applications.

Illustrated here is a constellation consisting of three private synchronizers and apps operated by participant nodes “OP” for “Operator”, each providing a prerequisite piece for the DvP transaction discussed in the introduction. A cash registry provides payment functionality, a fund tokenization app an asset to be purchased, and a trading app a marketplace where a Buyer and a Seller meet and agree a trade. The public *Global Synchronizer* common to all three operators and the two trading participants allows the DvP transaction to be coordinated amongst those five nodes.

2.2 Identity

Akin to blockchain systems, user generated cryptographic public key fingerprints form the foundational layer of Canton's identity system. Unlike most other blockchains, however, Canton adds a layer of abstraction to identity in order to make it easier to manage keys (to rotate or revoke them, for example), to reuse keys (one key for many wallets), or to set up complex policies (multi-sig, or read-only access to wallets). This abstraction layer is known as the *topology ledger* and next to managing participant node and synchronizer identities and keys, the topology ledger also manages Canton's abstract on-ledger identity called a *party*. In most situations, the best mental model is to simply identify keys, participant nodes, and parties one to one and think of that triple as a wallet or address. But to give a faithful account of Canton's ledger model and consensus in sections 2.3 and 2.4, understanding the technical distinction between parties and participants and their relation to keys is of value. That's what this section covers. An in depth treatment of Canton's identity management is available in the documentation[36].

Namespaces form the roots of trust in Canton's topology ledger. A namespace is identified by the fingerprint of the public key of a private/public key pair called a *namespace root key.* By default, participant nodes are in one to one correspondence with namespaces and generate their namespace root key during initialization. Canton also has the ability to bootstrap multisig namespaces where there is no single controlling key, but that's beyond this paper.

Note in particular that like in other public networks, this setup ensures that there is no single trusted root of trust, but each entity represents themselves by generating their own identity and own root of trust. Participants must verify each others' identities.

Identities are tuples name::namespace and thus rigidly linked to a namespace. By default, a participant node will generate its own identity in the namespace controlled by it during initialization, using a participant-generated name in the name segment.

Topology changes like creating a new identity are made through *topology transactions* signed by the namespace root key and distributed through synchronizers. Topology transactions follow a simple ledger model akin to certificate chains, and the resulting ledger of topology transactions is the *topology ledger*. By default, a participant node will broadcast the topology changes and state for their namespace to all synchronizers that they are connected to. A good mental model for this paper is that all topology transactions are globally known and consistent on all synchronizers. It is possible to maintain different topology states facing different synchronizers, and it is possible to delegate signing of topology transactions to *namespace intermediate keys*, but this has no impact on the rest of this paper so will not be discussed further.

[36] Canton Identity Architecture Documentation, https://docs.daml.com/canton/architecture/identity.html

Participants from a ledger viewpoint are identities with several associated key pairs, two of which are important for understanding the protocol. The first key pair is the *signing key*, which the participant uses to authenticate and sign any messages as part of the consensus protocol. The second is the *encryption key* used to encrypt messages sent from one node to another through the synchronizer. The public keys for both pairs are publicly known as part of the topology state.

Parties are abstract identities best thought of as equivalent to “wallets” or “addresses” in other blockchains. This is the notion of identity used in the Canton ledger model and smart contracts. They are mapped to participants in a *hosting relationship*, which is established through topology transactions. By default, a party created by a participant is in the same namespace as the participant, and automatically mapped to the participant in *submission* mode, meaning the participant can authorize transactions for that party. In this case, the signing key of the participant is equivalent to the private key of a Bitcoin address. It is also possible to host parties in *observation* mode, meaning read-only, or *confirmation* mode meaning participation in consensus on behalf of that party, but no authority to submit transactions. Parties can also be hosted on multiple nodes at the same time, with N-of-M schemes for consensus. This flexibility of modes enables simple topologies similar to Bitcoin and Ethereum or more complex ones often required by enterprise best practices and security requirements.

2.3 Canton Ledger Model

The Daml documentation[37] covers the Daml ledger model in depth. As the core subject of this paper is programming Canton using languages other than Daml, it will be referred to as the Canton ledger model here, and introduced to the needed level of detail.

Extended unspent transaction output (eUTXO) ledger models form the core of many blockchain systems and applications that have privacy and confidentiality at their core. To name just a few: Corda[38], ZCash[39], Aztec[40], Aleo[41], Zeto[42]. Canton is no exception. This pattern has good reasons that would go beyond the scope of this paper to discuss in depth. But in short, the UTXO ledger model, as opposed to Ethereum’s account based ledger model, breaks up the ledger state into small immutable pieces - UTXOs. This makes it easier to break up ledger state and distribute it on a need to know basis, allowing for confidentiality. It also makes it easier to do collision detection, enabling double spend protection while only partially knowing the ledger state.

Ledger state in a eUTXO ledger is a set of UTXOs, which are immutable pieces of data. UTXOs are *created* as the output of a transaction, used as input to other transactions while they are *active*, and are eventually *archived* or *spent* by some transaction. One of

[37] Daml documentation on the Daml Ledger Model, https://docs.daml.com/concepts/ledger-model/index.html
[38] Corda: An introduction, Brown / Carlyle / Grigg/ Hearn, 2016, https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf
[39] ZCash Protocol Specification, Bowe/Hornby/Wilcox, 2018-2024, https://zips.z.cash/protocol/protocol.pdf
[40] An introduction to Aztec, Blog, Andrews, 2019, https://aztec.network/blog/an-introduction-to-aztec
[41] Aleo documentation, https://developer.aleo.org/concepts/public_private/#aleo-state-storage
[42] Zeto readme, https://github.com/hyperledger-labs/zeto

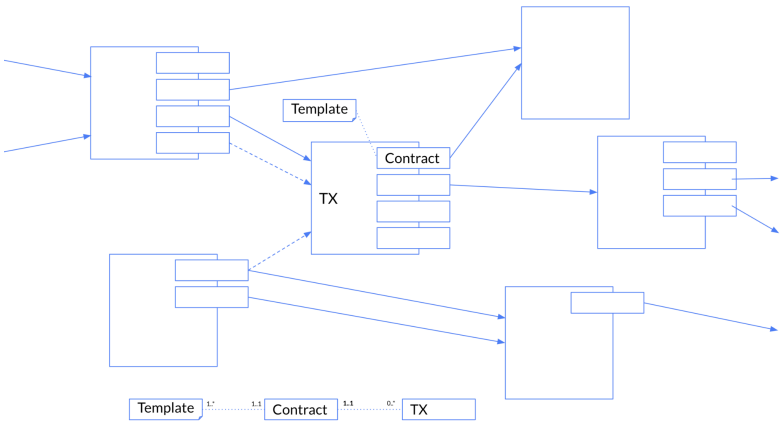
the distinguishing features of eUTXO as opposed to Bitcoin's UTXO is that eUTXOs are *typed* and can hold arbitrary data according to their type.

An eUTXO transaction consists first and foremost of a possibly empty set of UTXOs as inputs. It consumes some of these inputs and creates new outputs, which start in an unspent state.

Transaction validity is where eUTXO ledgers differ the most from each other as well as from plain UTXO. Validity is a pretty broad concept and can span everything from a transaction having the right cryptographic signatures to satisfying some transaction constraint like the sum of input Bitcoin being equal to the sum of output Bitcoin plus some transaction fees. All UTXO ledgers have some way to define what transactions are valid by attaching code to UTXOs. In Bitcoin, this is done by attaching a script[43] to each UTXO, which acts as a predicate for any transaction that tries to use the UTXO as input and evaluates whether the use is valid. If the attached code is rich enough this code is called *smart contracts*, and it can thus be said that even Bitcoin supports smart contracts[44], it is just not geared towards it. This model of attaching code to UTXOs is shared by many eUTXO models, for example Corda[45], but it is not the only way as section 2.5 will discuss.

An eUTXO ledger is a labeled directed acyclic graph (DAG) of eUTXO transactions where the transactions are nodes, and the edges are from transactions that output a UTXO to transactions that use the UTXO as input. The edges are labeled with UTXO identifiers, and whether they are used in a consuming or non-consuming fashion (called Reference States in Corda[46], and Reference Inputs in Cardano[47], and Data inputs in Ergo[48]). Each UTXO can appear on at most one consuming edge, a key property equivalent to not allowing double spends. In Canton, this graph is called the *causality graph*.

Canton's eUTXO model, illustrated here, at the surface looks exactly the same, with spent/unspent UTXOs being called active/archived *contracts.* Contracts are typed by being linked to contract templates, or *templates* for short, which determine both their data type and the shape of transactions that use the contract. We will get back to templates in Section 2.5.



[43] Script on the Bitcoin wiki, https://en.bitcoin.it/wiki/Script
[44] Contracts on the Bitcoin wiki, https://en.bitcoin.it/wiki/Contract
[45] Smart Contracts in the Corda documentation, https://docs.r3.com/en/platform/corda/5.2/developing-applications/ledger/smart-contracts.html
[46] Reference States, Corda Documentation, https://docs.r3.com/en/platform/corda/5.2/developing-applications/ledger/transactions.html#reference-states
[47] CIP-31 Reference Inputs, Cardano CIPs, https://cips.cardano.org/cip/CIP-31
[48] Data Inputs, Ergo Documentation, https://docs.ergoplatform.com/dev/protocol/tx/read-only-inputs/

Contract Ids are Canton's addressing scheme for transaction outputs. To express the graph pictured above, both transactions and transaction outputs need globally unique identifiers called contract Ids and Transaction Ids, respectively. This paper won't discuss how they are computed in Canton. Since Canton's ledger truly is a DAG, not a linear order, and has privacy requirements, it's not as simple as in Bitcoin, where UTXOs are addressed through transaction id and index. But an important property of contract Ids is that they are *authenticated*, meaning that the contract Id cryptographically commits to the data on the contract.

A hierarchical transaction structure enriches Canton's ledger model beyond typical eUTXO models. A transaction is a list of *actions*, which come in different types covered below. Some actions, specifically *exercise actions*, can have *consequences,* which are themselves a list of actions or correspondingly a transaction. This endows transactions with a tree, or more precisely a forest structure where nodes are actions and edges are via consequence relationships. A *sub-transaction* of a transaction is obtained by taking a set of actions from the forest, together with all their (transitive) consequences. As will become clear later, a good way of thinking about actions is as calls to procedures, the tree structure as a call graph, and a sub-transaction as a set of calls within that graph plus the call graph generated by them.

Actions come in several types, not all of which are covered here. The omitted actions concern transaction rollbacks, contract keys, or are subtle variants of exercises none of which add to the discussion in this paper. The important two actions to consider are:

Create actions which, as the name suggests, create new contracts. In eUTXO terms, they correspond to transaction outputs. They create new UTXOs. Create actions can hold arbitrary data, and additionally specify a non-empty set of parties called the *signatories*. The latter signifies that these parties have agreed to the contract and as such have some sort of obligation. Analogous to how it's best to think of smart contracts as persistent scripts, it's best to think of the signatories as the parties that have authorized the creation of some state and that jointly maintain it.

Exercise actions are named to convey the idea of a set of parties exercising a right on a contract. Those parties are specified by an exercise action and are called the *actors*. In persistent script terms, it's best to think of exercise actions as calls to a script or procedure on a contract. Such calls are always on an active contract, called the *input contract*, and this has the same meaning as in eUTXO. Also just like in eUTXO, the exercise action has a *kind*, which is consuming or nonconsuming, which determines whether the input contract is spent. Exercise actions have consequences, which are further actions.

Every exercise action has an input contract and every create action an output contract. The actions are said to be *on* the respective contract.

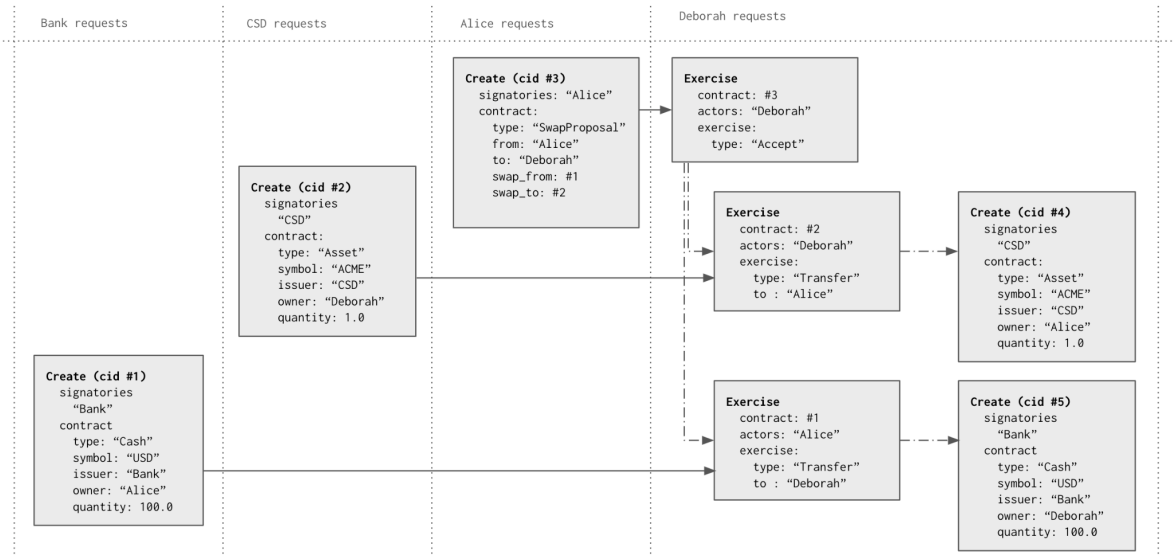
Ledger commits are transactions *requested* or *submitted* by one or more parties, the requester(s), as defined in 2.2 Identity.

A Canton ledger is a list of ledger commits, similar to how the Bitcoin ledger is the list of all blocks or equivalently all committed Bitcoin transactions. The following EBNF-like grammar summarizes the structure as treated in this paper:

```
Action      ::= 'Create' party+ contract
              | 'Exercise' party+ contract exercise Kind Transaction
Transaction ::= Action*
Kind        ::= 'Consuming' | 'NonConsuming'
Commit      ::= party+ Transaction
Ledger      ::= Commit*
```

This image illustrates a simple Canton ledger with four commits, leading up to an atomic DvP.

1. Bank issues some cash to Alice
2. CSD issues an asset to Deborah
3. Alice creates a proposal for the DvP
4. Deborah accepts and settles



Solid arrows represent inputs, as in the above UTXO illustration. Dash-dot arrows represent consequences.

Canton's validity model is subdivided into three sub-properties: *Consistency*, *Conformance*, and *Authorization*. Put simply, consistency says that there are no double spends, conformance says that all transactions adhere to the rules encoded in smart contracts, and authorization means that all parties that need to authorize an action have authorized it directly or indirectly.

Conformance and authorization impose important restrictions on the shape of transactions, which are relevant to language design, as a language ought to facilitate the creation of valid transactions. They are therefore defined more precisely below.

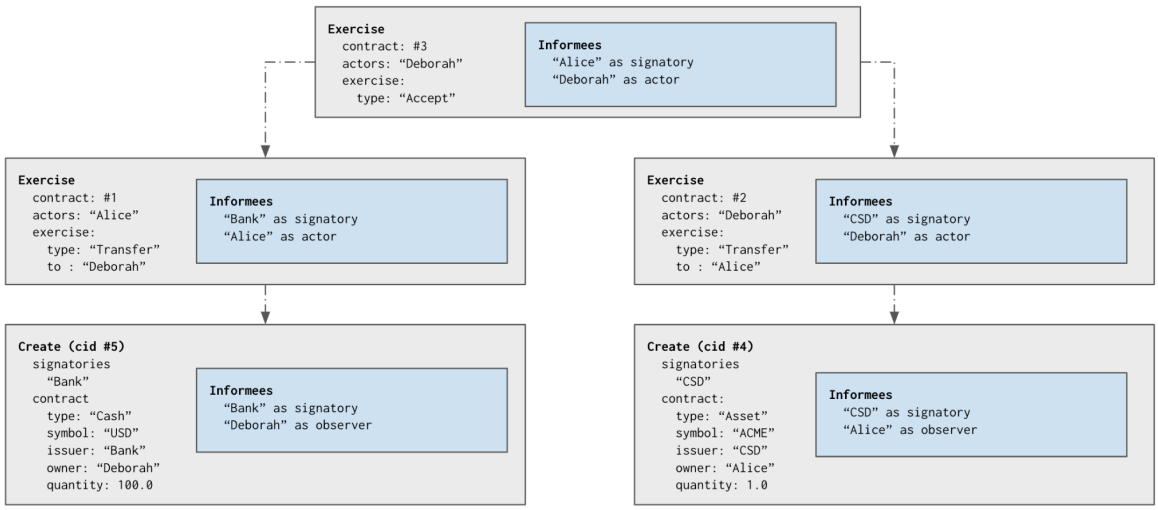
Conformance restricts which actions are allowed in the ledger. The set of all allowed actions is called the *contract model*. For example, a contract model suitable for the above example may specify that all create actions have a payload that is either an Iou signed by both specified parties, a PaintOffer, or a PaintAgree each with some given argument types. And it may specify that the only allowed exercise on an Iou is a consuming “transfer” specifying a new owner, with the current and new owners as actors, and with the creation of a new Iou owned by the new owner as its only consequence. An important property of any conformance model is that it is closed under sub-transactions, meaning if a contract model allows a transaction T, then it also allows any subtransaction S of T.

Authorization restricts who may request which actions, and puts further constraints on the contract model. The *required authorizers* of an action are the signatories for a create action, and the actors for an exercise action. The *authorizers* of an exercise action are the signatories of the input contract plus the actors. The authorization rule says that:

For top level actions in a ledger commit, the required authorizers are a subset of the requesters.

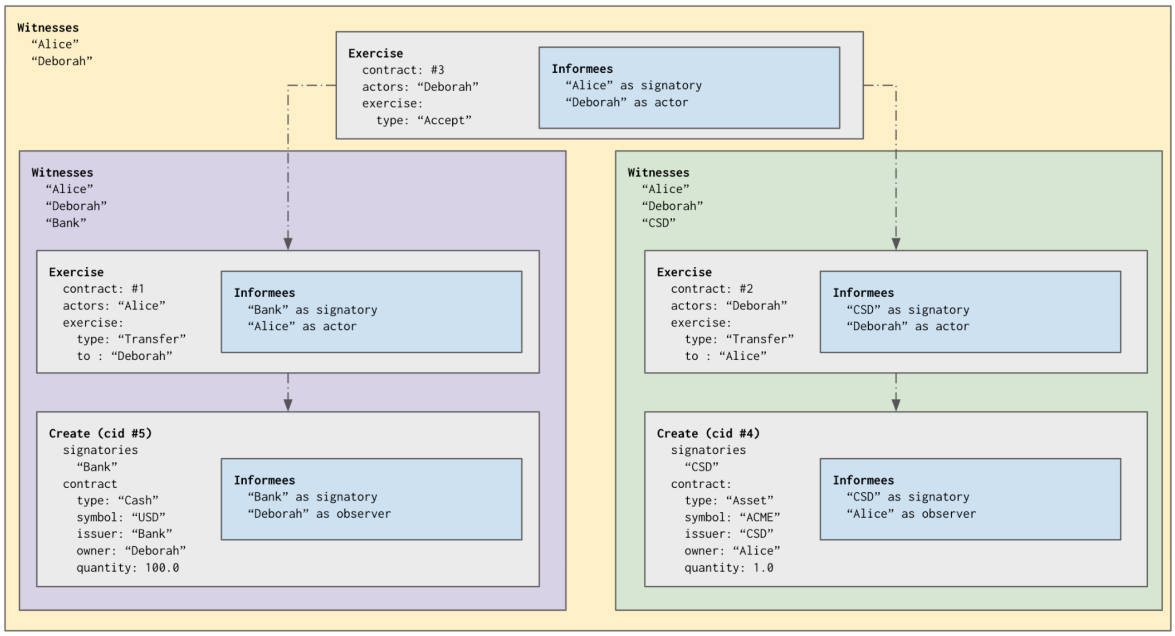
The authorizers of a parent action need to include all required authorizers of a child action, meaning a direct consequence.

Privacy and confidentiality are achieved by specifying the *projection* of the ledger that a party gets to see. Each action has a specified set of *informees*. The informees are the union of the action authorizers and an extra set of parties called the *observers,* which the Daml language section 3.1 will discuss further. A party’s projection of a transaction is the minimal subtransaction containing the actions on which that party is an informee. A party’s projection of the ledger is the set of commits for which the party’s projection is non-empty. A party is said to *witness* an action if it is in their projection. The difference between informee and witness can be explained as follows: A party may witness only the creation of a contract between other parties, but that doesn't mean that the party will be informed about further actions on that contract. An informee is guaranteed to see any consuming exercise on a contract for which they are the informee.



This image shows the informees in the final “swap” transaction of the above DvP example, under the assumption that the owners of Cash and Asset type contracts are observers. Note that this is a simplification from more realistic setups where the owner is likely a signatory. Since both observers and signatories are informees of a contract, this makes no difference for privacy considerations.

The below image shows the same transaction once more, highlighting the different *views* for the four involved parties. Witnesses are only ever added when going down the tree, so the *view tree* is a simpler tree than the tree of actions. It will be discussed further in the next section on Canton’s consensus.



The definitions guarantee that an informee of a create action are also an informee on any consuming exercise action on the resulting contract. The projection of the ledger can

therefore sensibly be applied to the state consisting of active contracts as well. A party's ledger state is the set of active contracts for which they were an informee of the create action. Participant nodes store and index their projection of the ledger and ledger state.

The Local Ledger[49] of a party is its projection of the global Canton ledger. Typically no party is privy to the whole global ledger. The rules and definitions of the Canton ledger model ensure that the party projections of a valid Canton ledger are also valid Canton ledgers, and that consistent, valid local ledgers can be merged into a valid global Canton ledger.

2.4 Canton Consensus

Canton's consensus follows a typical UTXO pattern, with extensions associated with the two tier architecture (2.1), the identity abstractions (2.2), and transaction structure, authorization and privacy (2.3).

Commands are submitted by users to a node, corresponding to *requesting* a commit in the ledger model. As such, the user submits the authorizing parties and the root actions of the commit. The node must either host the authorizing parties in submission mode, or the user must appropriately sign the transaction later in the process. In the case of exercise actions, they specify the input contract Id only, and not the input contract as a whole, or the consequences of the action. If the interpretation step requires contracts that the node does not have in its local storage, the user may also supplement the commands with additional contract information via a functionality called *explicit disclosure*[50].

Interpretation is the transformation of the commands to a conformant and well authorized ledger commit in the sense of the ledger model. To do so, it draws on the parties' ledger state available to the participant, as well as any additional user-supplied contracts. Interpretation can fail, but does so deterministically. If it does succeed, the resulting commit is the *unique* conformant transaction that could be interpreted from those commands. This is a slight oversimplification as in practice the user may omit some type information that the participant then fills in using heuristics, but for the purpose of this paper, this assumption on the contract model is valid and further restricts the nature of allowable contract models: If two conformant exercise actions match in all but their consequences, then their consequences also match. The output of interpretation is a Daml-LF transaction, discussed further in section 3.2.

Blinding is the process by which the submitting node decomposes the commit into its views and encrypts them. Witness parties are mapped to witness participants using hosting relationships in the topology state. The tree of actions is then transformed into a tree of *views* such that the witness participants on all actions within a view are the same and the tree structure is compatible, meaning the actions in a sub-view are (transitive) consequences of the actions in the parent view. The views are encrypted using participant

[49] Local Ledgers, Daml Documentation, https://docs.daml.com/concepts/local-ledger.html#local-ledgers
[50] Explicit Contract Disclosure, https://docs.daml.com/app-dev/explicit-contract-disclosure.html

encryption keys such that exactly the participants that host the witness parties of a view can decrypt them. The participant knows all the public keys and hosting relationships thanks to the shared topology state discussed in 2.2.

A parallel *confirmation tree* is generated that specifies which views need to be confirmed by which participants. In effect, this specifies which participant quorum is needed on behalf of each party that authorizes an action within the view. This is used in the stakeholder-based proof of authority algorithm described further below.

In the simple case where parties are fully hosted by single participants, the confirmation policy is that every participant hosting an authorizer of an action within the view needs to confirm the view.

The views, including their matching confirmation subtrees are assembled into a big multicast message called a *confirmation request*. This message also addresses the confirmation tree as a whole to a subcomponent of the synchronizer called the *mediator*. The confirmation request (or more precisely the transaction id, which is a root hash of the view tree) is signed by the participant or user depending on whether the requesting parties are hosted in submission mode, or not.

Sequencing involves the submitting participant sending the whole confirmation request to the synchronizer, or specifically a subcomponent called the *sequencer*. The sequencer records the confirmation request in a single total ordering with all other messages it processes, and then makes the individual views available to the addressees in that ordering. The sequencer acts as a multicast messaging queue with guaranteed consistent ordering and delivery. It doesn't know anything about the payload of the message. From the sequencer's perspective, every sequencing request is a batch of multiple envelopes with CC and BCC addresses containing encrypted messages.

Validation is done by the participant nodes that receive their views of the confirmation request. Since they all get the confirmation request in consistent order, they all have consistent ledger state projections and can thus validate their projections consistently and deterministically. A contract is either active to all its signatories or to none, for example. Consistency checking therefore boils down to each participant checking that all input contracts for which they host a signatory party, are active as specified.

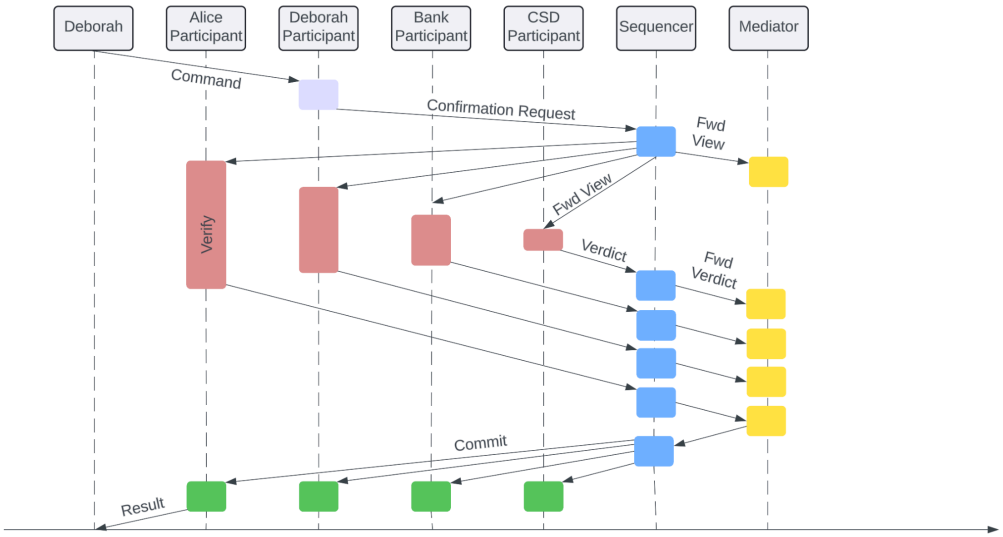
Thanks to the contract model being closed under subtransactions, and having unique conformant transactions for each action, conformance and authorization checking can be done by taking the root nodes of the view, and *reinterpreting* and *reblinding* them. The node goes through the interpretation and blinding steps again and checks that it gets the same views. For this, the node doesn't rely on its stored contracts, but the inputs specified on the actions themselves. This ensures that witnesses can fully perform conformance and authorization checks for their views, and is safe since contract Ids commit to their data, and have already been checked for activeness.

In a non-faulty, non-malicious setting the *only* failures are write contention (two conflicting transactions attempting to spend the same UTXOs, or with topology changes), as well as timeouts. This is a desirable property for a DLT system since failing transactions are costly.

Confirmation is the core of the stakeholder based proof of authority algorithm. Nodes specified in the confirmation tree send their *verdict* on their view to the mediator via the sequencer. Verdicts are either approvals or rejections.

Mediation involves the mediator aggregating all the confirmations, and evaluating them according to the confirmation tree. Once enough confirmations have been received to correspond to positive verdicts from every authorizing party, the mediator addresses a *mediator confirmation* in a single multicast message through the sequencer to all involved participants. This has similar privacy properties to the sequencing of the confirmation request above.

Commit of the ledger happens with the recordation of the mediator confirmation on the sequencer. In logical sequencer time, this is at the same point in time for all participant nodes involved. They apply the transaction and state changes to their local data stores as they process this message.



As an end result, we have an atomic commit with low trust and high confidentiality. The synchronizer learns the shape of the transaction via the confirmation tree, the full set of involved participants, and has access to encrypted messages. This is not dissimilar to an internet ISP that learns all servers who are communicated with and how much, and also has access to all encrypted packages, but is unable to decrypt the payloads.

Smart contracts come in twice here, during interpretation and validation. The important extra requirement this consensus protocol imposes on the language stack is the uniqueness property described under interpretation. In the mental model of exercise actions being procedure calls, this property is equivalent to *determinism* of the procedures.

2.5 Smart Contracts in Canton

In some sense, the role of smart contracts in Canton is simple. They are a *constructive* specification of the contract model. Constructive in the sense that they are not a predicate like scripts in Bitcoin, or contracts in Corda, which each put constraints on transactions that are built using separate code paths. Instead, the conformant transactions are exactly those that can be built by the smart contract code, and the same code can be used for validation.

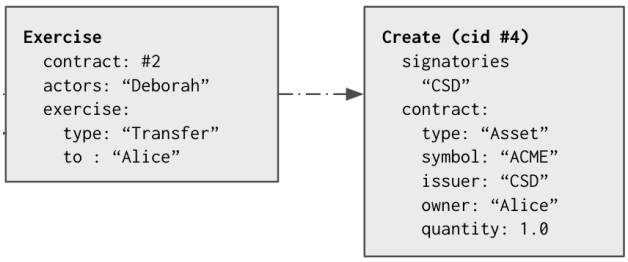
This makes development conceptually easier in two ways.

Firstly it's aligned with the prevalent *imperative* thinking in transactional systems and programming. If one were to implement an Iou registry in memory or using SQL, it is natural to ask the question "What should a call to transfer do?", rather than thinking in terms of transaction constraints like "What invariants should any transaction on my system conserve?". This is one of the great benefits of Ethereum's account based system with replicated state machines. Mutating state through imperative calls is strongly aligned with many developers' way of thinking. Canton gets close to that in a UTXO model.

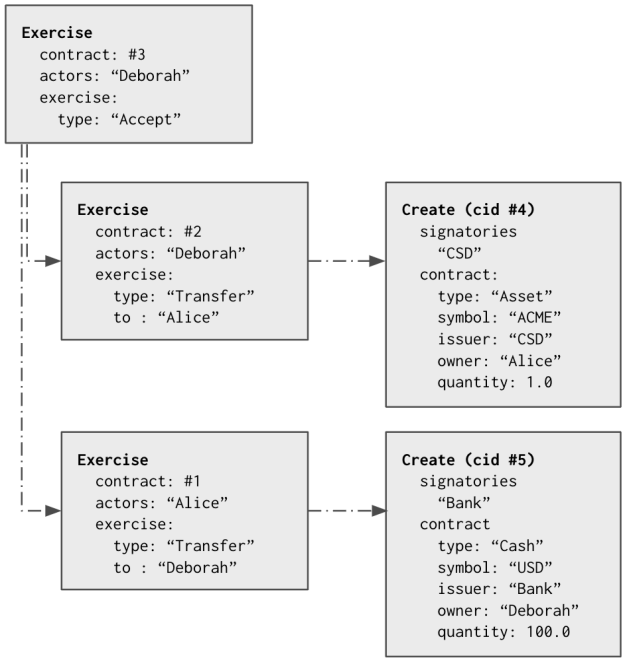
Secondly, it means there is only one code path. Many other eUTXO systems have one code path to construct transactions, and a second to validate them. The construction is often considered "off-ledger" even though it is just as important for a functioning system as the validation.

Composition is also aided by the constructive nature of the specification and the property that the contract model is closed under subtransactions. Imagine a naive way of implementing a "transfer" using a classic eUTXO model by saying "transactions are valid exactly if they have one input and one output with equal amounts, and the transaction is signed by the owner of the input". Now a developer that wants to implement an atomic swap is stuck, as such a transaction has two inputs and two outputs. The developer of the original transfer has to prepare for this by expressing validity with great care: "transactions involving this token as an input are valid if the sum of amounts across all tokens of matching type is equal to the sum of the amounts on the output tokens of this type, and the transaction is signed by all owners on any of the inputs".

Contrast constructive eUTXO transactions as in Canton as illustrated here. Developer one specifies that a transfer is conformant by writing a procedure that creates consuming "transfer" exercise on an asset with the creation of a new asset as its sole consequence. Developer two extends the contract model by constructing swaps. They write a procedure that calls the "transfer" procedure for the delivery and payment assets respectively. This way of composing aligns with the imperative mindset, and the closure under subtransactions. This is where the correspondence between exercise actions and procedure calls comes back into play. From a developer perspective, composition is via procedure calls to previously specified procedures.



From a ledger model perspective, composition is achieved by specifying new conformant actions that have previously specified conformant actions as consequences.



Packages are units of smart contract code in Canton used to extend the contract model. They are identified in a globally unique way through their package Id, which is a hash of their “binary” representation which is distributed. Every create and exercise action references an identifier within a package, either a *template* or a *choice*.

Templates are smart contract code corresponding to create actions. By analogy, one can think of templates as equivalent to a class definition in an object oriented language. Templates consist of a data type for the *create arguments* of a contract, as well as functions which compute contract metadata and invariants like signatories and observers from those arguments. These functions are called during (re-)interpretation when a contract of this type is written or read.

Due to this dual purpose of templates it also sometimes makes sense to distinguish the *contract arguments*, the data payload that the contract stores, and the contract itself, which is the result of instantiating the template with given arguments.

Choices are code corresponding to exercise actions. Similar to the above analogy, one can think of choices as equivalent to class methods in an object oriented language. They take *exercise arguments* and compute exercise actions, consisting both of metadata like the actors as well as the consequences. The code specifically computing the consequences is called the *choice body*. A choice corresponds exactly to the “transfer” and “swap” procedures in the illustrative example above.

Data Serializability is needed for both contract and exercise arguments. Canton uses gRPC[51] APIs throughout, so all serializable data needs to be easily mappable to Protocol Buffer (protobuf)[52] messages. Contract and exercise arguments form an essential part of commands that users send *to* the API, of the Canton ledger model transactions that a user *gets back* from the API, and of the views as part of the confirmation request that is distributed through the synchronizer.

[51] gRPC website, https://grpc.io/
[52] Protocol Buffers (protobuf) website, https://protobuf.dev/

Package Vetting is the process by which parties (technically currently participants) say which packages they are willing to participate in. The *vetting state* is part of the topology state and changed through topology transactions. New packages allow the application of new templates and choices to existing contracts, which makes it possible to upgrade smart contracts. However, since a party authorizes any choice on a contract that they are signatory on, adding choices needs mutual agreement from all signatories. This is achieved through vetting.

In conclusion, at a very high level, we need the following from a smart contract language stack for Canton:

A *surface language* that makes it easy for a developer to express both the serializable data types of choice and exercise arguments, as well as the procedures corresponding to templates and choices. The procedure that computes the exercise consequences must be able to reference templates and choices from other (dependency) packages to allow for composability.

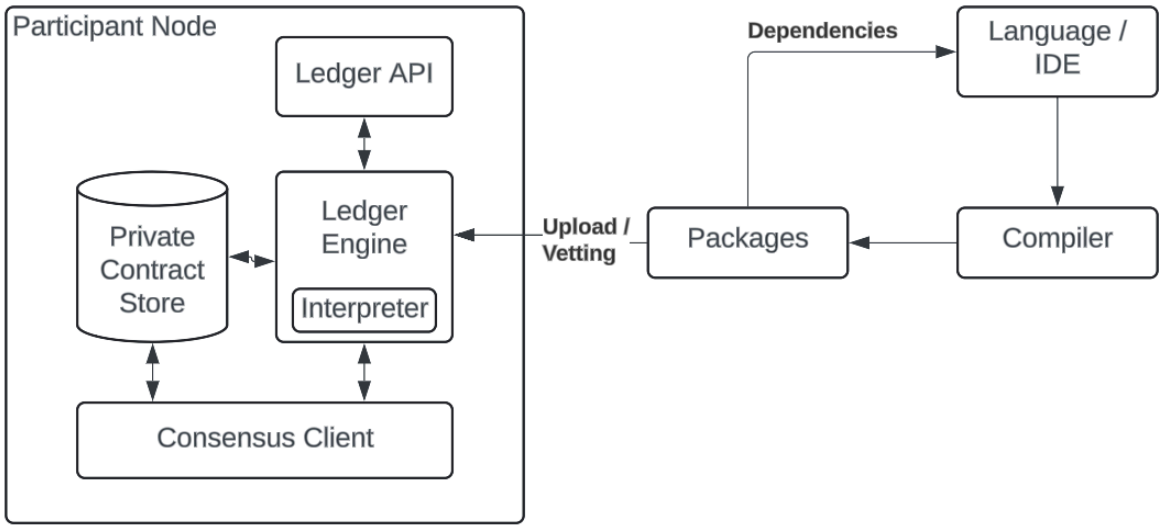
A *compiler* that turns the surface language into a package containing some sort of executable bytecode or intermediary language, including template and choice definitions.

An *interpreter/VM* that runs the bytecode.

The interpreter is hosted by a *canton ledger engine* (or just *ledger engine*), which is in charge of all (re-)interpretation. The canton ledger engine connects to API, private contract store (PCS), and consensus client, and hosts the interpreters. It is called by the API or consensus client to perform a (re-)interpretation. To do so, it acts as a builder for Daml-LF transactions. For any command (or action) that it needs to interpret, it resolves the package and any inputs from the PCS, and then calls the appropriate interpreter to compute metadata or consequences. When the interpreter wants to append a consequence, it calls a host function provided by the ledger engine, which passes back control from the interpreter to the ledger engine. The ledger engine can now repeat this recursively, maintaining a call stack of choices. This illustrates again how the tree of actions corresponds closely to a call graph of smart contract functions. The ledger engine is also responsible for checking authorization on the fly.

The contract between ledger engine and interpreter is that computations by the interpreter need to be fully deterministic. The resulting transactions are thus conformant by construction. Determinism includes certain safety properties: Isolation between parallel transitions. Isolation from the system, and similar.

The *consensus client* can independently take care of consistency checks, and perform blinding, sequencing, confirmation, and commit.



The interaction between the ledger engine and the smart contract code running in the interpreter can be expressed as a series of function evaluations to either *pure* or *update* expressions. The computation of contract and exercise metadata (e.g. signatories) are pure expressions. They are side-effect free and can neither read, nor update ledger state beyond the arguments supplied.

signatory expression: (Contract Arguments => [Party])

The choice body is an update expression. The signature of such an expression is similar to a pure one, but as part of the computation, it can both read and update ledger state by calling back to the engine to trigger consequences.

choice body expression: ((Contract Arguments, Exercise Arguments) => Return Value)

The engine maintains ledger state starting from an initial state S against which interpretation is run. As update expressions create consequences, a partial transaction T is built up, and the engine maintains the resulting ledger state S'. Any read operations in an update expression are performed against the S' obtained by applying the partial transaction up to that point to S:

S' = S + T

The final resulting Daml transaction T is then projected onto Canton views and submitted as a confirmation request for validation and will only be applied if the interpretation result based on S is invariant with respect to the actual ledger state at sequencing time S'' of the confirmation request. This means that if two transactions race to modify related ledger state, one of them will fail, which may require a resubmission of the command. This

resubmission might result in a different transaction due to the fact that the ledger state changed in the meantime. The advantage of this implementation is the ability to run all expensive computations and read-write operations in parallel, while restricting the sequential computation part to in-memory conflict checking, resulting in throughputs well above 8k actions per single participant node (Daml Enterprise 2.8.0).

3 Daml

The design of the Daml smart contract language was driven by a number of factors laid out in the Daml whitepaper[53]. In summary, it allows developers to express the construction rules of a contract model for Canton concisely, safely, and rapidly as it provides domain specific primitives for templates, choices, and all their “metadata” like signatories and observers.
All this needs to be solved under a hard constraint for determinism and safety as covered in section 2.5. In the years before ecosystems like Cosmos (CosmWasm[54]) and Polkadot (Substrate[55]) in 2019, this was a hard problem to solve. Smart contract blockchains took one of three approaches, illustrated here by example.

Hyperledger Fabric is designed to simply deal with non-determinism[56]. This has a great advantage that it can support any general purpose language. It has the disadvantage that any non-determinism effectively leads to forks. The chain may contain transactions that some participants deem invalid. Furthermore, running vanilla general purpose code is inherently *unsafe* as it can access the system and perform I/O. Sandboxing in containers is a way to alleviate this, but realistically, this design restricts smart contracts to trusted code.

Corda attempted to modify Java, a specific existing general purpose language, to be deterministic. This required the development of a deterministic Java Virtual Machine (dJVM)[57]. This has the same advantage of using a general purpose language as Fabric’s approach, and removes some of the disadvantages. But this approach has a high cost and difficulty. After years of development, the dJVM was removed from Corda in 2023[58]. Secure EcmaScript (SES)[59] is another initiative in this direction.

Daml, similar to Solidity for Ethereum, represents the third way by developing an entirely custom stack, consisting of custom surface language tailor made for the ledger model, and a fully deterministic, safe, custom engine that exposes a domain specific set of host functions.

As previously discussed, Canton’s ledger model was historically called the Daml ledger model as this approach allows tight cohesion between language and ledger model.

[53] Daml: A Smart Contract Language for Securely Automating Real-World Multi-Party Business Workflows, Bernauer et al, 2019, https://arxiv.org/abs/2303.03749
[54] CosmWasm History, https://cosmwasm.com/home/history/
[55] Preparing for Polkadot’s launch with substrate, Blog, 2019, https://polkadot.com/blog/preparing-for-polkadots-launch-with-substrate
[56] Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains, Elli Androulaki et al, 2018, https://arxiv.org/abs/1801.10228
[57] dJVM code repo, https://github.com/corda/djvm
[58] Corda 4.11 release notes, https://docs.r3.com/en/platform/corda/4.11/enterprise/release-notes-enterprise.html#djvm-removal
[59] Secure EcmaScript, GitHub, https://github.com/tc39/proposal-ses

Daml's stack as presented in section 2.5 consists of the surface language, which will be introduced by example in section 3.1, a compiler based on the Glasgow Haskell Compiler (GHC)[60], an intermediary language, Daml-LF, that is output by a custom compiler backend, and the Daml engine. Section 3.2 discusses the technical stack further, in particular to highlight integration points and modifications for the integration of additional languages.

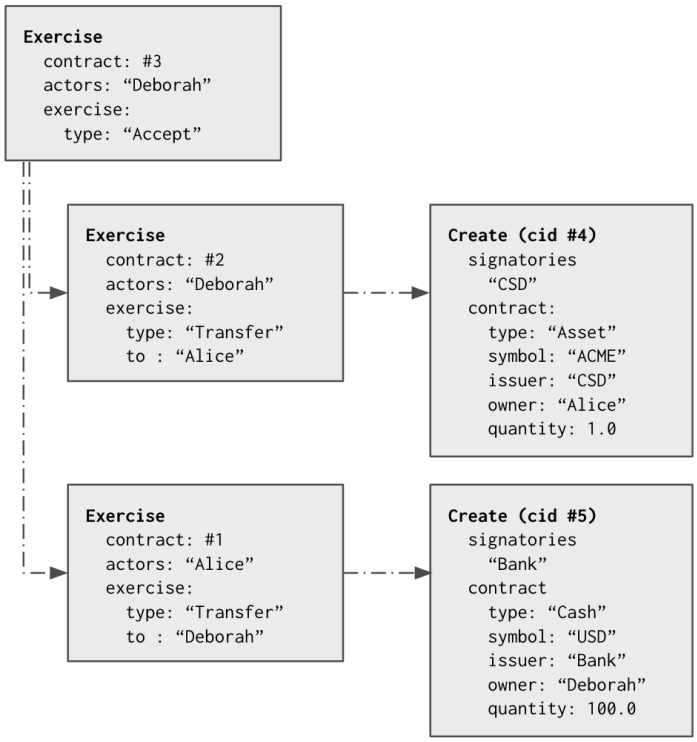
Haskell was used as a basis for Daml both for the ease with which domain specific languages can be embedded in GHC, as well as for Haskell's compatibility with the determinism requirement. Pure functional languages are specifically designed to isolate side effects, which are the primary source of non-determinism.

3.1 Surface Language

The example shown here is the simple* token and swap model corresponding to the examples used in the Canton ledger model section (2.3).

There are two types of creates in this transaction, meaning two templates are needed: Asset and SwapOffer. Illustrated here is the Asset.

**A more realistic example with both the issuers and owners as signatories requires additional preparatory steps leading up to the DvP, and would make it challenging to compactly illustrate a valid ledger leading up to the atomic swap transaction.*



[60] GHC homepage, https://www.haskell.org/ghc/

```
-- Asset is a template with a data type carrying the issuer and owner
-- which are Parties as defined in section 2.2.
template Asset
 with
   issuer : Party
   owner  : Party
   symbol : Text
   quantity : Decimal
 where
-- The diagram tells us that the issuer should be the signatory.
   signatory issuer
-- The owner should have assets in their private contract store, so
-- they are added as an extra informee to create and consuming archive
-- events. This is done through the observer metadata which works
-- in analogy to signatory.
-- Typically the owner would also be a signatory but for illustration
-- purposes we model the owner as an observer in this example
   observer owner

-- We have a single choice Transfer. Its sole argument is the new owner,
-- and it returns the reference to the newly created output.
   choice Transfer : ContractId Asset
     with
       newOwner : Party
-- The actors for the resulting exercise are specified through the
-- `controller` metadata on the choice.
     controller owner
-- Consequences are computed in the choice body. In this case, the sole
-- consequence is the creation of a new Asset with the new owner.
     do create this with
          owner = newOwner
```

As we can see from this, Daml *encapsulates* choices in templates. The choices that apply to a contract created through a template are expressed on the template itself. This gives Daml an almost object oriented character, where templates are classes, and choices are methods.

The Cash template looks identical to Asset so this example uses Asset for both. The SwapPropsosal is shown here:

```
template SwapProposal
 with
   from : Party
   to : Party
```

```
  swap_from : ContractId Asset
   swap_to : ContractId Asset
where
  signatory from
  observer to

  choice Accept : (ContractId Asset, ContractId Asset)
    controller to
    do
      -- Check that ownerships match.
      -- `fetch` is another ledger action similar to nonconsuming
      -- exercises which reads active contracts.
    fromAsset <- fetch swap_from
      toAsset <- fetch swap_to
      assert (fromAsset.owner == from)
      assert (toAsset.owner == to)

      swap_from' <- exercise swap_from Transfer with newOwner = to
      swap_to' <- exercise swap_to Transfer with newOwner = from

      return (swap_to', swap_from')
```

Daml's IDE has the ability to simulate Canton ledgers in a script, used here to demonstrate a ledger leading up to a swap:

```
simulate : Script ()
simulate = script do
 -- Topology transactions to set up identities.
 -- Scripts assume parties are single-hosted
 -- on distinct participants.
 -- Package vetting is not needed in scripts.
 alice <- allocateParty "Alice"
 deborah <- allocateParty "Deborah"
 bank <- allocateParty "Bank"
 csd <- allocateParty "CSD"

 -- Submit first commit to create cash.
 -- `submit party` corresponds to a commit with
 -- `party` as top level authorizer.
 -- Note this is two commands in a single commit.
 swap_from <- submit bank do
  createCmd Asset with
      issuer = bank
```

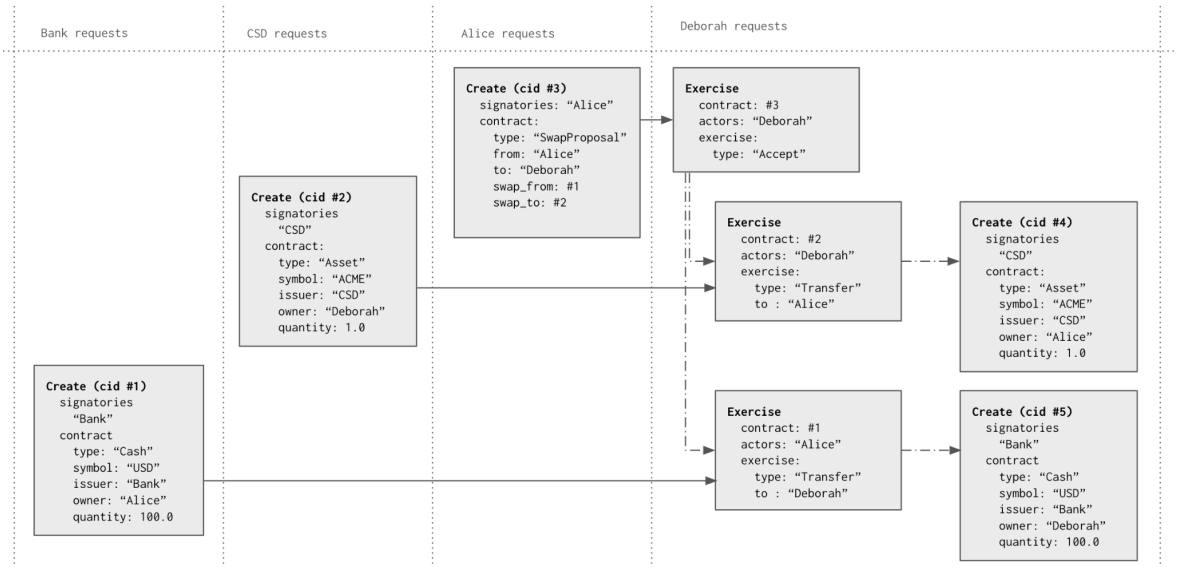
```
  owner = alice
    symbol = "USD"
    quantity = 100.0
 swap_to <- submit csd do
  createCmd Asset with
    issuer = csd
    owner = deborah
    symbol = "ACME"
    quantity = 1.0

-- Make a SwapOffer via another commit.
offerCid <- submit alice do
  createCmd SwapProposal with
    from = alice
    to = deborah
    swap_from = swap_from
    swap_to = swap_to

-- Alice needs to give Deborah details about her asset
-- in order for her to be able to interpret the swap.
Some disclosedFromAsset <- queryDisclosure alice swap_from

-- Swap in a third commit.
-- Bob needs to augment his private store with information
-- about Alice's asset.
submitWithDisclosures deborah [disclosedFromAsset] do
  exerciseCmd offerCid Accept
return ()
```

The resulting ledger is exactly the one from 2.3:



The above presents the Daml language purely as a tool for expressing the Canton ledger model. A different mental model helpful for many learners of Daml is an analogy with databases. The below uses postgreSQL syntax to illustrate.

A template's type corresponds to a table.

```
template Asset
  with
    issuer : Party
    owner  : Party
    symbol : Text
    quantity : Decimal
```

```
CREATE TABLE Asset(
  id        SERIAL PRIMARY KEY,
  issuer    TEXT  NOT NULL,
  owner     TEXT  NOT NULL,
  symbol    TEXT  NOT NULL,
  quantity  REAL  NOT NULL);
```

Creates correspond to inserts.

```
submit bank do
  createCmd Asset with
    issuer = bank
    owner = alice
    symbol = "USD"
    quantity = 100.0
```

```
INSERT INTO Asset (issuer, owner, symbol, quantity)
VALUES ('Bank', 'Alice', 'USD', 100.0)
```

Contracts/UTXOs correspond to table rows. Archiving/spending a contract corresponds to deleting rows. There is no equivalent of an update operation on a database.

Signatories correspond to row based write access controls. Observers correspond to row based read access controls.

```
signatory issuer
```

```
ALTER TABLE Asset ENABLE ROW LEVEL SECURITY;

CREATE POLICY asset_issuers ON Asset
    USING (issuer = current_user);
```

```
observer owner
```

```
CREATE POLICY asset_owners ON Asset
    FOR SELECT
    USING (owner = current_user);
```

Choices correspond to stored procedures with atomic bodies, exercises to calls.

```
choice Transfer : ()
  with
    newOwner : Party
  controller owner
do
  create this with
    owner = newOwner
```

```
CREATE PROCEDURE transfer(asset_id INT, new_owner TEXT)
LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO Asset (issuer, owner, symbol, quantity)
    (SELECT issuer, new_owner, symbol, quantity
      FROM Asset WHERE id = asset_id);
  DELETE FROM Asset WHERE id = asset_id;
  END;
```

```
exercise swap_from Transfer
  with newOwner = to
```

```
CALL transfer(swap_from, newOwner);
```

There are no exact analogies for choice controllers, and Daml's authority transfers from the signatories of an input contract of an exercise to any calls made within its choice body. Daml templates and choices can also be seen as an API specification since creates and exercises are the primary actions a user can take via Canton's ledger API.

3.2 Language Stack

The damlc compiler compiles Daml source code to an intermediary language called *Daml-LF*, short for *Daml Ledger Fragment*. Damlc is implemented using a fork of GHC (a Haskell compiler). GHC type checks and compiles the Daml source code into terms based on System-F_ω[61], an extension of the simply typed lambda calculus. GHC is configured to compile Daml code deterministically - this allows participants to verify that uploaded Daml-LF has been derived from a given or known set of Daml sources.

Daml-LF[62], when viewed as a term rewriting system, is strongly normalizing. Hence, Daml-LF is interpreted deterministically to a unique resulting value, regardless of how that evaluation proceeds.

Daml Packages are the result of compiling Daml source files into module-scoped Daml-LF expressions, and are stored in *Dalf files*.

[61] System FC, as implemented in GHC, Simon Peyton Jones, https://gitlab.haskell.org/ghc/ghc/-/blob/master/docs/core-spec/core-spec.pdf
[62] Daml-LF specification, https://github.com/digital-asset/daml/blob/main/sdk/daml-lf/spec/daml-lf-2.rst

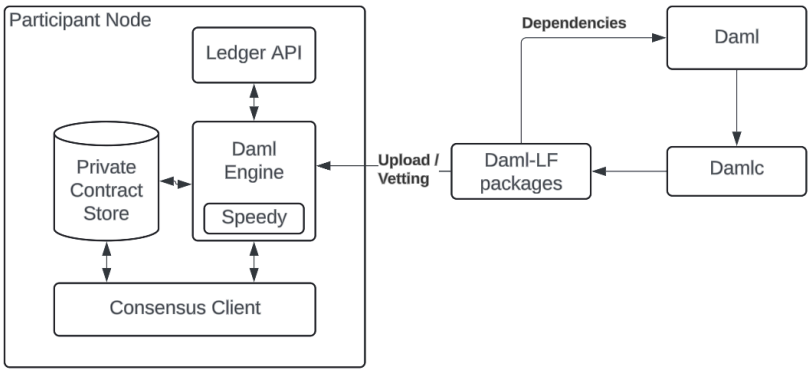
Multiple Dalf files may be zipped together, along with some metainformation, to form a *Dar file*. Dar files may then be uploaded to Canton participants for vetting and use by the participant's Daml engine during, for example, submission, reinterpretation or replay.

Dar files are usually in one to one correspondence with packages and contain the original compiled Daml source modules, along with *all* transitive compiled module dependencies.

Package IDs are the *unique* hashes (e.g. SHA256) of the Dalf file contents (i.e. the compiled Daml modules). As Daml source code is deterministically compiled, package IDs may always be verified by anyone who has access to the Daml sources.

Speedy is Canton's Daml-LF interpreter. It is an efficient CEK machine[63], interpreting the lfpackage terms using a (non-serializable) internal value model, ultimately producing a transaction. The speedy interpreter is pure and needs to be hosted in an engine that connects it to storage and network.

The Daml Engine performs roughly the role of the Canton ledger engine described in section 2.5. It connects Speedy to ledger API, PCS, and consensus components. The main delta between the idealized model shown in 2.5 and the implementation is that it is Speedy which builds transactions, not the engine. This is one of the changes section 4 addresses for the purpose of hosting other interpreters. As the interpreter and engine build the transaction tree, authorization checks are performed and a contract state machine is used to validate all potential updates of the participant's active contract store.



Daml Engine Builtin Commands are used in Daml-LF to define updating ledger operations as side effects. These builtin commands allow contracts to be created and fetched. They also allow sub-transactions to be opened as contract choices start their evaluation and closed when that evaluation successfully terminates. Should a sub-transaction's evaluation fail in any manner, the sub-transaction can be closed by aborting it. A finalized transaction will have no open sub-transactions - i.e. they will all have been closed or aborted. This paper refers to the collection of Daml-LF and engine builtin commands as the *engine host interface*.

[63] Distilling abstract machines, Accattoli, Barenbaum , Mazza, 2014, https://dl.acm.org/doi/10.1145/2628136.2628154

When contract choices are exercised, it is important to ensure that sub-transaction evaluation is isolated. In other words, separate sub-transactions should not interact with each other except via ledger state. The Daml engine achieves this as:

- Daml-LF expressions have no access to state other than via triggering actions,
- and the Daml engine shares no internal state when evaluating Daml-LF expressions.

Daml Engine Resource Safety and Security are ensured by breaking up evaluation into a series of partial evaluation stages. This allows the Daml engine to be regularly interrupted allowing Canton participants to control the engine's compute budget. Interruptions occur after a fixed number of evaluation iterations by Speedy.

In addition, the Daml engine bounds the depth to which recursive calls may be made when evaluating Daml-LF expressions. This avoids JVM stack overflows when evaluating Daml-LF expressions.

To ensure that Canton participant memory resources are resilient to scenarios where the Daml engine has aggressive memory allocation demands, Canton uses package vetting. With package vetting, Canton participants only run packages that they have actively permissioned. Thus, no participant is forced to run untrusted code.

The Daml engine implementation targets the JVM to aid portability and to ensure that its implementation is decoupled from the host operating system.

The Daml-LF transactions[64] produced by interpreter and engine and as introduced in 2.4, are protobuf messages that can be processed further by the consensus client. The transaction schema captures the tree structure of Canton ledger model actions with all its metadata. Serializability of Daml-LF transactions is key as its nodes form an integral part of the confirmation request views which are transmitted over the wire.

Daml-LF values[65] are the serializable constant terms that can appear in transactions. It's a protobuf AST that describes:

- data values such as unit values, booleans, integers, numerics, strings, dates and timestamps
- structured data such as optionals, sorted maps, sorted lists and records (sorted by field name)
- and ledger oriented data values such as contract IDs and parties.

LF values are used to encode:

- arguments for template constructor functions
- arguments for template choice functions
- and result values returned from *exercising* a contract's choice function.

[64] Daml-LF transaction specification, https://github.com/digital-asset/daml/blob/main/sdk/daml-lf/transaction/src/main/protobuf/com/digitalasset/daml/lf/transaction.proto
[65] Daml-LF value specification, https://github.com/digital-asset/daml/blob/main/sdk/daml-lf/transaction/src/main/protobuf/com/digitalasset/daml/lf/value.proto

Decoupling Daml from Canton and allowing additional languages to be supported requires a clean separation between the Daml-specific parts of this stack and those that need to be shared by all supported languages for interoperability and integration. Section 4 will demonstrate by example how engine, LF transactions, and LF values can be decoupled from Speedy and LF terms in such a way that a second language and interpreter can be hosted with full Daml interoperability.

4 Wasm-based Smart Contracts in Canton

WebAssembly (abbreviated Wasm) is a "safe, portable, low-level code format"[66], similar to JVM ByteCode or Daml-LF. Wasm is used in both web and non-web use cases to run untrusted code in a strong sandbox[67]. Two of its properties make Wasm particularly useful for executing smart contracts:

- *Deterministic Execution*: Applications execute deterministically with a limited set of well-defined exceptions[68]. This enables the reinterpretation and validation of transactions produced by smart contracts where the output of a smart contract must be consistent across all nodes executing it.

- *Controlled Side-Effects*: By default all applications are pure without any side-effects. Side-effects are strictly controlled via a set of host functions, which depend on the use case.

Wasm code is organized in *modules*[69] which include imports and exports. Imports include functions that the Wasm code requires to be provided by an instance of the Wasm engine, which could be satisfied with host functions or by loading another module. Exports include functions that are provided by the Wasm code and can be called by the host (or another module that imports those exported functions).

Wasm's safety and structural properties, the many languages that can compile to Wasm, and its community support make it an ideal candidate for an additional interpreter next to Speedy.
This section presents initial work[70] to integrate the Chicory Wasm runtime[71] in Canton and program Canton native contracts using high level languages like Rust and AssemblyScript.

Section 4.1 picks up from sections 2.5 and 3.2 to show how engine, LF-transactions, and LF-values can be decoupled from Speedy and LF-terms such that a second interpreter can be hosted in Canton. A key property that's demonstrated here is the ability to make smart contract calls between Daml contracts and Wasm contracts. Section 4.1 uses Rust as a surface language to generate Wasm, and interacts with the

[66] Introduction to the WebAssembly Specification, https://webassembly.github.io/spec/core/intro/introduction.html
[67] WebAssembly Design Documentation on Security, https://github.com/WebAssembly/design/blob/main/Security.md
[68] WebAssembly Documentation on nondeterminism, https://github.com/WebAssembly/design/blob/main/Nondeterminism.md
[69] Webassembly Design Documentation on Modules, https://github.com/WebAssembly/design/blob/main/Modules.md
[70] Wasm Engine integration in Canton, Github, https://github.com/digital-asset/daml/pull/20159
[71] Chickory Wasm runtime, Github, https://github.com/dylibso/chicory

engine host interface at a low level, explicitly manipulating the bytestrings used for data interchange between engine and interpreter.

Section 4.2 explains which of the aspects explained in section 4.1 can be abstracted away using libraries and code generation to illustrate what a finished programming experience for Wasm-based Canton smart contracts could look like.

4.1 Wasm Interpreter in Canton

The Chicory Wasm runtime was chosen for integration in Canton both because it is JVM based, but also because it supports lightweight invocation of many interpreter instances which is needed to prevent any global state affecting determinism.

As laid out in sections 2.5 and 3.2, there are three primary interactions between engine and interpreter. Firstly, the interpreter needs to be able to call engine host functions for consequences, most importantly to create another contract, or to call a choice on another contract. Secondly, the engine needs to be able to call the interpreter for pure expressions used for contract and exercise metadata. Thirdly, the engine needs to be able to call the interpreter for update expressions, most notably for choice bodies.

Host and Guest are used in this section, respectively, to refer to the engine, and the code running in the Wasm interpreter.

The engine host functions for ledger consequences are specified in a Scala trait:

```
trait WasmHostFunctions {
  def logInfo(msg: String): Unit
  def createContract(templateCons: Ref.TypeConRef, args: LfValue): LfValue.ContractId
  def exerciseChoice(
     templateId: Ref.TypeConRef,
     contractId: LfValue.ContractId,
     choiceName: Ref.ChoiceName,
     choiceArg: LfValue,
  ): LfValue
}
```

Pure and update expression evaluation distinctions are enforced through two sets of host function implementations, one for pure evaluation (PureWasmHostFunctions), and one for update expressions (UpdateWasmHostFunctions). Calling the host functions in the pure instance simply results in an error.

```
object PureWasmHostFunctions {
 import WasmUtils._
 import internal.WasmRunnerHostFunctions._
```

```
    val createContractFunc: WasmHostFunction =
     wasmFunction("createContract", 2, WasmValueResultType) { _ =>
       throw new RuntimeException(
         "Host functions can not be called from pure WASM exported functions: createContract"
       )
     }
    val exerciseChoiceFunc: WasmHostFunction =
     wasmFunction("exerciseChoice", 4, WasmValueResultType) { _ =>
       throw new RuntimeException(
         "Host functions can not be called from pure WASM exported functions: exerciseChoice"
       )
     }
}
```

Data transfer between host and guest is done through memory pointers. WebAssembly only supports primitive types[72] and relies on the application to build complex types, such as strings or bytestrings, on top of those primitive types.

In this section all data exchange is done via bytestrings, represented as a tuple (pointer: i32, length: i32). i32 is used as the Wasm memory is currently limited to 32-bit addressing.

Host functions are limited to returning a single value, therefore a pointer: i32 to the bytestring tuple is returned. Thus all data interchange between engine and interpreter is done by passing memory pointers to bytestrings between the host and guest. All this is taken care of by the wasmFunction construct seen above.

```
 private[wasm] def wasmFunction(name: String, numOfParams: Int, returnType:
Option[WasmValueType])(
    lambda: Array[ByteString] => ByteString
): WasmHostFunction = {
  new WasmHostFunction(
    (instance: WasmInstance, args: Array[WasmValue]) => {
      require(args.length == numOfParams)

      copyByteString(
        lambda((0 until numOfParams).map(copyWasmValues(args, _)(instance)).toArray)
      )(instance)
    },
    "env",
    name,
    (0 until numOfParams).flatMap(_ => WasmValueParameterType).asJava,
    returnType.toList.asJava,
  )
}
```

[72] WebAssembly Documentation on Types, https://webassembly.github.io/spec/core/syntax/types.html

Explicit memory management between host and guest is required by the Wasm runtime. This is done inside the copyByteString and copyWasmValues functions above.

```
private[wasm] val WasmValueParameterType = List(WasmValueType.I32)
private[wasm] val WasmUnitResultType = None
private[wasm] val WasmValueResultType = Some(WasmValueType.I32)
private[wasm] val i32Size = WasmValueType.I32.size()

private[wasm] def copyWasmValue(values: Array[WasmValue])(implicit
    instance: WasmInstance
): ByteString = {
  copyWasmValues(values, 0)
}

private[wasm] def copyWasmValues(values: Array[WasmValue], index: Int)(implicit
    instance: WasmInstance
): ByteString = {
  require(0 <= index && index < values.length)

  val byteStringPtr = values(index).asInt()
  val ptr = instance.memory().readI32(byteStringPtr)
  val size = instance.memory().readI32(byteStringPtr + i32Size)

  ByteString.copyFrom(
    instance.memory().readBytes(ptr.asInt(), size.asInt())
  )
}

private[wasm] def copyByteString(
    value: ByteString
)(implicit instance: WasmInstance): Array[WasmValue] = {
  copyByteArray(value.toByteArray)
}

private[wasm] def copyByteArray(
    value: Array[Byte]
)(implicit instance: WasmInstance): Array[WasmValue] = {
  if (value.isEmpty) {
   Array.empty
  } else {
    val alloc = instance.export("alloc")
    val valuePtr = alloc.apply(WasmValue.i32(value.length))(0).asInt
    val byteStringPtr = alloc.apply(WasmValue.i32(2 * i32Size))(0).asInt
```

```
    instance.memory().write(valuePtr, value)
    instance.memory().writeI32(byteStringPtr, valuePtr)
    instance.memory().writeI32(byteStringPtr + i32Size, value.length)

    Array(WasmValue.i32(byteStringPtr))
  }
}
```

In order to obtain a valid pointer in the guest's memory, the host needs access to memory allocation and deallocation methods by the guest. The guest module provides an implementation of memory management aligned with the guest's language specific memory constraints and exports those functions to the host. Such a call is visible above in instance.export("alloc").

For example, in Rust, memory allocation and deallocation methods are implemented as the following, which needs to exclude the allocated memory from Rust's ownership-based memory manager, otherwise the memory will be deallocated before the host can use it.

```
pub fn alloc(len: usize) -> *mut u8 {
    let mut buf = Vec::with_capacity(len);
    let ptr = buf.as_mut_ptr();
    std::mem::forget(buf);

    return ptr;
}

pub unsafe fn dealloc(ptr: *mut u8, size: usize) {
    let data = Vec::from_raw_parts(ptr, size, size);
    std::mem::drop(data);
}
```

The guest code represents the bytestring tuple of pointer and length as:

```
#[repr(C, packed)]
#[allow(non_snake_case)]
pub struct ByteString {
    pub ptr: *const u8,
    pub size: usize,
}
```

Deterministic execution of Wasm based smart contracts is largely supported out of the box. The execution of a WebAssembly application is deterministic with a few well-defined

exceptions outlined in the Wasm documentation on nondeterminism[73]. Features that lead to nondeterminism such as threads or SIMD are not enabled for the execution of smart contracts. Nondeterminism stemming from floating point arithmetic and NaN bitwise representations can be mitigated either by NaN canonicalization by the engine or through code instrumentation.

However, with projections and sub-transactions in the Canton ledger model there is another potential source of non-determinism during reinterpretation. Wasm supports mutable global variables, which in principle allows for the passing of information from one action to another outside of ledger state and exercise arguments. Different participants may have different actions as their entry-point of their views of the transaction. Consequently, participants execute only part of the original Wasm application that produced the entire transaction in the first place. This could result in different values in global variables, and thus non-deterministic behavior.

Therefore, it is crucial that the execution of every pure or update expression is done in a Wasm engine instance that does not have any global state from prior execution. This is accomplished by creating a new Wasm interpreter for every call both during interpretation and re-interpretation. Depending on pure or update use, the interpreter is instantiated with different host function implementations.

```
private def PureWasmInstance(): WasmInstance = {
  val imports = new WasmHostImports(
    Array[WasmHostFunction](
      ...
      PureWasmHostFunctions.createContractFunc,
      PureWasmHostFunctions.exerciseChoiceFunc,
    )
  )

  WasmModule.builder(wasmExpr.module.toByteArray).withHostImports(imports).build().instantiate()
 }

 private def UpdateWasmInstance(): WasmInstance = {
  val imports = new WasmHostImports(
    Array[WasmHostFunction](
      ...
      createContractFunc,
      exerciseChoiceFunc,
    )
  )

  WasmModule.builder(wasmExpr.module.toByteArray).withHostImports(imports).build().instantiate()
}
```

[73] WebAssembly Documentation on nondeterminism, https://github.com/WebAssembly/design/blob/main/Nondeterminism.md

Wasm based templates and choices must expose their pure and update expressions using a naming convention. For example, a template SimpleTemplate must expose the signatories expression as SimpleTemplate_signatories so that the engine can invoke it correctly.

The engine can now invoke Wasm-based smart contract code for either pure or update expressions, exchange arguments and return values, and handle calls to host functions for consequences in update expressions. An example invocation is shown here:

```
 val signatories = wasmTemplateSignatoriesFunction
  (templateName, txVersion)(argsV)(PureWasmInstance())
```

wasmTemplateSignatoriesFunction takes care of the naming convention and return value unwrapping.

```
private[wasm] def wasmTemplateSignatoriesFunction(
    templateName: String,
    txVersion: TransactionVersion,
)(
    contractArg: LfValue
)(implicit instance: WasmInstance): Set[Party] = {
  wasmTemplateFunction(s"${templateName}_signatories", txVersion)(contractArg)
   match {
    case LfValue.ValueList(values) =>
      values
        .map {
          case LfValue.ValueParty(party) =>
            party
          case _ => ...
        }
        .iterator
        .toSet
    case _ => ...
  }
}
```

wasmTemplateFunction takes care of bytestring (de-)serialization and passing, and invokes the Wasm runtime.

```
private[wasm] def wasmTemplateFunction(
    functionName: String,
    txVersion: TransactionVersion,
```

```
)(contractArg: LfValue)(implicit instance: WasmInstance): LfValue = {
  val function = instance.export(functionName)
  val contractArgPtr = copyByteString(
    LfValueCoder
      .encodeValue(txVersion, contractArg)
      .fold(err => throw new RuntimeException(err.toString), identity)
  )
  val resultPtr = function.apply(contractArgPtr.head)
  try {
    if (resultPtr.nonEmpty) {
      LfValueCoder
        .decodeValue(txVersion, copyWasmValue(resultPtr))
        .fold(err => throw new RuntimeException(err.toString), identity)
    } else {
      LfValue.ValueUnit
    }
  } finally {
    deallocByteString(contractArgPtr.head)
    deallocByteString(resultPtr.head)
  }
}
}
```

Writing Canton smart contracts using Wasm requires:

- access to the Wasm host functions for consequences
- export guest functions according to the naming convention to the host
- no use of any other host functions, in particular no dependency on WASI (Wasm System Interface)[74]

Rust is a good choice for low-level integration due to its mature Wasm support and full control over import and export of functions.
Host functions are declared as external functions by a *low-level library*, which will result in entries of the guest's import table. For example, externally defined function for the createContract host function:

```
 extern {

      #[allow(non_snake_case)]
         pub fn createContract<'a>(templateTyCon: &'a ByteString, arg: &'a ByteString) -> &'a
ByteString;

  }
```

[74] WASI, https://wasi.dev/

The other engine host functions are similarly defined as external functions.

Guest exported functions for memory allocation and deallocation as shown in *Explicit memory management* are also provided by the low-level library.

A mid-level ledger update library provides typed access to the engine host functions and the low-level library uses the basic ByteString type in order to pass data between the host and guest and this is not convenient to use for a smart contract developer. For example, here is mid-level createContract function:

```
  pub fn createContract(templateTyCon: lf::Identifier, arg: lf::Value) -> lf::Value {
       unsafe {
         let templateTyConBytes = templateTyCon.write_to_bytes().unwrap();
         let argBytes = arg.write_to_bytes().unwrap();
          let templateTyConByteString = internal::ByteString { ptr: templateTyConBytes.as_ptr(),
size: templateTyConBytes.len() };
          let argByteString = internal::ByteString { ptr: argBytes.as_ptr(), size: argBytes.len()
};

                 let  contractIdByteString  =  internal::createContract(&templateTyConByteString,
&argByteString);

         return utils::to_Value(contractIdByteString.ptr, contractIdByteString.size);
       }
     }
```

This wrapper performs the necessary steps to convert LF identifiers and LF values into low-level bytestrings that can be moved across the host-guest boundary.

Similar wrapper functions exist for all the other low-level host functions to record ledger updates.

Traits for templates and choices defined in the mid-level library define what methods a smart contract developer has to provide for both contract templates and contract choices. The template trait in Rust specifies (amongst other things) the need for a list of associated choices and expressions for signatories and observers:

```
   pub trait Template<T> {
       fn new(arg: lf::Value) -> T;

       fn choices() -> HashMap<String, Box<dyn Choice>> {
               return HashMap::new();
```

```
  }

    #[allow(non_snake_case)]
    fn toLfValue(&self) -> lf::Value;

    fn signatories(arg: lf::Value) -> lf::Value;

    fn observers(arg: lf::Value) -> lf::Value {
            let mut result = lf::Value::new();
            let empty = lf::value::List::new();

            result.set_list(empty);

            return result; // lf::value::List<lf::value::Party>
    }
}
```

In full, the template trait captures the required functionality that needs to be implemented by a smart contract developer:

- Creating a new template instance given an LF value argument
- Return the template's identifier
- Return the choices defined for this template
- Convert the template arguments back to an LF value
- Optional preconditions that need to be satisfied by the template's arguments
- Metadata for the signatories, observers, key and maintainers

Similarly, a Choice trait captures:

- The kind, indicating if the choice is consuming or not
- The exercise function for the choice body that computes the consequences
- Metadata for the choice controllers, observers, and additional authorizers.

SimpleTemplate is an example of the implementation of the Template and Choice traits. It has a contract state that contains an owner party and counter integer:

```
pub struct SimpleTemplate {
    owner: String,
    count: i64,
}
```

The template is implemented as the following in Rust (excerpt):

```
impl ledger::api::Template<SimpleTemplate> for SimpleTemplate {
     fn new(arg: lf::Value) -> SimpleTemplate {
       // SimpleTemplates are created using paired (record) arguments
       let owner = ledger::utils::get_field(arg.clone(), 0).take_party();
       let count = ledger::utils::get_field(arg, 1).int64();

       return SimpleTemplate {
              owner: owner,
              count: count,
         };
     }

     fn choices() -> HashMap<String, Box<dyn ledger::api::Choice>> {
         let mut result = HashMap::new();

     result.insert(String::from("SimpleTemplate_increment"),
           Box::new(SimpleTemplate_increment) as Box<dyn ledger::api::Choice>);
         result.insert(String::from("SimpleTemplate_decrement"),
           Box::new(SimpleTemplate_decrement) as Box<dyn ledger::api::Choice>);

         return result;
     }

     #[allow(unused)]
     fn signatories(arg: lf::Value) -> lf::Value {
          let mut owner = ledger::utils::get_field(arg, 0);
          let mut result = lf::Value::new();
          let mut list = lf::value::List::new();

          list.elements = vec![owner];
          result.set_list(list);

          return result;
     }
```

An excerpt of the increment choice is shown here:

```
impl ledger::api::Choice for SimpleTemplate_increment {
  fn consuming(&self) -> lf::Value {
    let mut result = lf::Value::new();

    result.set_bool(true);

    return result;
  }
```

```
  fn exercise(&self, contractArg: lf::Value, choiceArg: lf::Value) -> lf::Value {
     assert!(choiceArg.has_unit());

     let contract = SimpleTemplate::new(contractArg);

     let updatedContract = SimpleTemplate {
         owner: contract.owner,
         count: contract.count + 1,
     };

     return ledger::api::createContract(SimpleTemplate::templateId(),
       updatedContract.toLfValue());
   }

   fn controllers(&self, contractArg: lf::Value, choiceArg: lf::Value) -> lf::Value {
     assert!(choiceArg.has_unit());

     let owner = ledger::utils::get_field(contractArg, 0);
     let mut result = lf::Value::new();
     let mut list = lf::value::List::new();

     list.elements = vec![owner];
     result.set_list(list);

     return result;
   }
 }
```

Exported guest functions for a template like SimpleTemplate need to follow the naming convention, and handle input and return values via pointers to ByteStrings.

For example, the signatory computation function for SimpleTemplate:

```
  impl templates::SimpleTemplate {
       #[no_mangle]
        pub unsafe fn SimpleTemplate_signatories(argPtr: *const ledger::internal::ByteString) ->
*mut ledger::internal::ByteString {
           use protobuf::Message;

           let arg = ledger::utils::to_Value((*argPtr).ptr, (*argPtr).size);
           let result = templates::SimpleTemplate::signatories(arg);
           let resultBytes = result.write_to_bytes().unwrap();
       let boxedResult = Box::new(ledger::internal::ByteString { ptr: resultBytes.as_ptr(), size:
resultBytes.len() });
```

```
        std::mem::forget(resultBytes);

        return Box::into_raw(boxedResult);
    }
}
```

In the current work, these low-level export functions are hand-written. Automatic generation with additional tooling is part of future work.

Cross-language smart contract calls are possible and demonstrated in the current work.

The host function for exerciseChoice of the engine dispatches the execution of the choice's body either to the Wasm interpreter for Wasm based templates or to Speedy engine for Daml-based templates. In both cases, the ledger updates are recorded in a partial transaction builder in the engine.

Contracts written in Rust and executed by the Wasm-based engine can create and call choices on contracts written in Daml. For this the Rust template must have access to the template id of the Daml template as well as the choice name. Similarly, Rust-written choice bodies can create contracts for Daml templates by providing the template id and contract arguments as an LF value.

The reverse direction will need extensions to Daml in order to reference foreign templates and choices by id and name in similar fashion.

Cross-interpreter interoperability is generally possible because data exchanged between Wasm and Daml for template arguments or choice arguments are always in the existing form of LF values. The engine takes care of transaction building and authorization checks. It does not matter how updates are produced as long as they are properly authorized.

In the current work, the template ids, choice's names and arguments are hand-written in the Rust code. A standardized way of expressing template and choice interfaces across languages would enable universal code generation to provide typed cross-language calls to choices and contract creation.

4.2 High-level Language support

Based on the Wasm interpreter and low- to mid-level Rust work in 4.1, this section covers the general building blocks required in a surface language for smart contract development. The goal is to remain as close as possible to a language's standard compiler, IDE, development workflow, language syntax and programming model. Access to the Canton ledger model and concepts for smart contracts are provided in the form of libraries.

A suitable surface language must have tooling support for Wasm, i.e., to compile down to Wasm bytecode. To work with the engine's host interface, it must have the ability to

define and import host functions, the ability to define an arbitrary number of exported guest functions, and little or no dependency on other host functions such as WASI. As part of interacting with the engine, it must also have good support for protobuf (de-)serialization in order to work with Daml-LF.

Low- and mid-level libraries as demonstrated in section 4.1 take care of host function imports, guest memory management, bytestring representations and pointer passing, and protobuf (de-)serialization to and from LF values. The results are mid-level wrappers for templates, choices, and their corresponding host functions (e.g. create and exercise actions).

AssemblyScript is used here to illustrate just how different the result can look like between two Wasm-targeting languages like Rust and AssemblyScript, but also to illustrate a highly accessible high level language experience.

The template trait and createContract mid-level function in Rust are represented in AssemblyScript by a template class with a fully implemented create method. The call to internal.createContract refers to the low-level library.

```
export class Template {
   private _arg: LfValue;
    constructor(arg: LfValue) {
     this._arg = arg;
   }
   arg(): LfValue {
     return this._arg;
   }
   create<T>(): Contract<T> {
     let templateIdByteStr = internal.ByteString.fromProtobufIdentifier(
       templateId<T>().toProtobuf(),
     );
     let argByteStr = internal.ByteString.fromProtobuf(this.arg().toProtobuf());
     templateIdByteStr.alloc();
     argByteStr.alloc();
     let contractId = LfValueContractId.fromProtobuf(
       internal.ByteString.fromI32(
         internal.createContract(
           templateIdByteStr.heapPtr(),
           argByteStr.heapPtr(),
         ),
       ).toProtobuf(),
     );
     argByteStr.dealloc();
     templateIdByteStr.dealloc();
 }
```

A high-level language experience needs to go further by providing *type-safe* and *convenient* access to host functions, as well as a concise way to specify templates and choices.

Ledger-specific wrapper types represent the native types in Daml-LF values on top of language primitive types. For example, Party as a wrapper for strings:

```
class LfValueParty extends LfValue {
   private _party: string;
   …
}
```

High level, typed, ledger update methods are created by wrapping values in parameterized classes like Contract<T> already seen as a return type above. Similarly, the class Choice<T, A, R> can be parameterized by template, argument, and return types allowing for fully typed interaction with the ledger.

```
export class Choice<T, A, R> {
   private _contractArg: T;
   ...

   exercise(arg: A): R { ... }
}
```

Templates and Choices are implemented by extending the template and choice classes.

```

export class SimpleTemplate extends api.Template {
   private owner: string;
   private count: i64;
    constructor(owner: string, count: i64) {
     super(toLfValue<SimpleTemplate>(owner, count));
     this.owner = owner;
     this.count = count;
   }

   signatories(): Set<string> {
     return new Set<string>().add(this.owner);
   }

   choices(): Map<string, SimpleTemplate_increment_closure> {
     return super
       .choices()
```

```
      .set(
      "SimpleTemplate_increment",
      new SimpleTemplate_increment(this),
      );
  }
}
```

```
class SimpleTemplate_increment extends api.ConsumingChoice<
 SimpleTemplate, i64, api.Contract<SimpleTemplate>
> {

…

 exercise(arg : i64): api.Contract<SimpleTemplate> {
   api.logInfo(
     `called AssemblyScript SimpleTemplate_increment(${n}) with count = ${count}`,
   );

   return new SimpleTemplate(_contractArg.owner, _contractArg.count + arg).create();
 }
}
```

Contracts are created and choices are exercised not by calling constructors of the Template classes or exercise functions on Choice objects, but through create() functions on template instances as seen above, or by calling the exercise<A>(choiceName : String, arg : A) function on Contract<T> instances which as per the above represent wrapped contract Ids. This is crucial to allow the engine to insert the appropriate actions in the partial transaction.

LF Value Encoding and Decoding should be taken care of for the developer. There are two approaches for this, both of which still need further investigation. The first is to support near-arbitrary protobuf messages as contract and choice arguments instead of the current LF value hierarchy. In that case, the language specific protobuf compiler and (de-)serializers could be used directly instead of doing any custom encoding.

Alternatively, code-generation could be used to generate the LF value codecs that are hand-written in the current work.

```
export function toLfValue<SimpleTemplate>(
  owner: string,
  count: i64,
 ): api.LfValue {
  return new api.LfValueRecord(
```

```
    new Map<string, api.LfValue>()
      .set("owner", new api.LfValueParty(owner))
      .set("count", new api.LfValueInt(count)),
 );
}
 export function fromLfValue<SimpleTemplate>(arg: api.LfValue): SimpleTemplate {
  if (isValidArg(arg)) {
    let owner = arg.map.entries[0].value.party;
    let count = arg.map.entries[1].value.int64;
    return new SimpleTemplate(owner, count);
  } else {
    throw new Error(
      `${arg} is an invalid contract argument type for SimpleTemplate`,
    );
  }
}
```

Code Generation would likely also be used to generate the guest exported functions that the host must call for each specific pure or update expression that it needs access to, for example, to execute the body of a choice or retrieve template authorization information.

Based on the template and choice implementations, the following set of guest exported functions are generated:

- TemplateName_(signatories|observers): takes template arguments as serialized LF value, returns the serialized LF value of a signatories/observers set in the form of a bytestring
- TemplateName_precond: takes the template arguments as serialized LF value and returns a boolean if the precondition is met.
- For each template choice
    - TemplateName_ChoiceName: takes the template and choice arguments as LF values serialized to bytestring, returns choice result as serialized LF value
    - TemplateName_ChoiceName_consuming: returns boolean
    - TemplateName_ChoiceName_(controllers|observers): takes template and choice arguments to return choice controllers/observers as serialized LF value set of parties.

Convenience for interoperability with Daml and other foreign language templates can be achieved by generating “facade” contract implementations, that call the appropriate create and exercise ledger updates using the package id and template identifiers of those externally-defined templates.

The package id of a Wasm module is computed as the hash of the Wasm module’s bytes representation, analogous to package ids of Daml packages. We need tooling to code generate the facade contract implementations based on the package ids, template arguments and choices.

To make this possible universally amongst many languages, a common interface description similar to Ethereum's application bytecode interface's (ABI) JSON encoding[75] is likely needed. An ABI description of the templates and choices would become a standard part of packages for all Canton-supported languages, and each language could generate the facades from that API description.

5 Solidity and EVM support in Canton

EVM compatibility is a much sought after property for public blockchains. Several of the largest public networks in market capitalization have launched EVM compatibility projects in the past few years: Solana[76], Cardano[77], Ripple[78], Polkadot[79], Near[80], Aptos[81]. Others have been designed for various degrees of EVM compatibility from the get go: BNB[82], Tron[83], Avalanche[84], Polygon[85].

The VeChain documentation[86] covers well why EVM compatibility matters for *Developer Adoption* and *Code Reusability* reasons. But it also illustrates a typical narrative on resulting *interoperability*, which is only partially true.

Developer Adoption is a common argument for using Java (targeting the JVM). If the majority of developers in the field are most qualified and most interested in working in Java, it makes sense to write apps in Java. Easier hiring, easier training, faster ramp up, faster time to market.

Solidity, the primary language in the EVM ecosystem, has overwhelming market share in DeFi with over 90% of Total Value Locked (TVL) at the time of writing[87]. If Canton Network is to encompass everything from traditional regulated finance to DeFi, being able to address DeFi developers with minimal friction is of clear value. In the regulated enterprise space, too, numerous companies have built on private Ethereum clients like Besu, and thus developed a community of Solidity developers embedded in financial institutions.

Code Reusability is the second strong argument. The introduction (section 1) argues that the way to unlock value from blockchain is to move assets and services to a common venue where they can interoperate through atomic transactions. There are already many venues, both in the form of public networks, and private permissioned enterprise networks. So the "move" in the above statement really means "mobilize" by virtue of making assets *multi-venue*, not "build from scratch".

If Canton Network is to act as a common venue for traditional, regulated finance assets and services, traditional assets already tokenized on blockchains, as well as DeFi, then it must make it as easy as possible for existing blockchain applications to either

[75]
[76] Neon EVM website, https://neonevm.org/
[77] Introducing the EVM Sidechain, IOHK, 2022, https://iohk.io/en/blog/posts/2022/07/06/introducing-the-cardano-evm-sidechain/
[78] XRPL Sidechain website, https://www.xrplevm.org/
[79] Frontier Github repo, https://github.com/polkadot-evm/frontier
[80] Aurora website, https://aurora.dev/
[81] ByteBabel website, https://pontem.network/bytebabel
[82] BNB Smart Chain introduction, https://docs.bnbchain.org/bnb-smart-chain/introduction/
[83] Differences from EVM, Tron documentation, https://developers.tron.network/v4.4.0/docs/vm-vs-evm
[84] Port an Ethereum dApp to Avalanche, Avalanche documentation, https://docs.avax.network/dapps/end-to-end/launch-ethereum-dapp
[85] Polygon PoS, Polygon documentation, https://docs.polygon.technology/pos/
[86] EVM Compatibility, VeChain documentation, https://docs.vechain.org/core-concepts/EVM compatibility
[87] DeFi Llama Languages, https://defillama.com/languages

move to Canton Network wholesale, or to add Canton Network as an additional venue supported by the app. If the app is built on the EVM stack, then EVM compatibility makes this significantly easier.

Interoperability with DeFi is often cited as one of the main reasons why EVM compatibility is important:

> ***Interoperability:*** *EVM compatibility enables different blockchain networks to communicate and interact with each other. This allows developers to build decentralized applications that can be used across multiple blockchain networks, which enhances the interoperability of the entire blockchain ecosystem.*
>
> Source: VeChain documentation

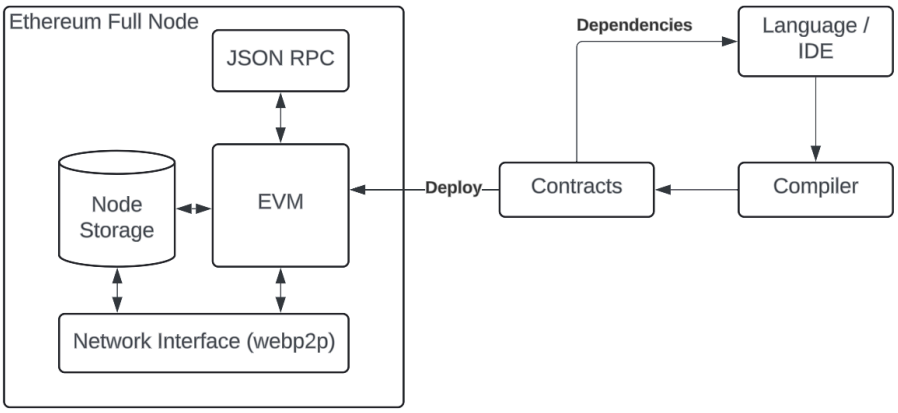
The first sentence is a common, but false narrative[88]. It amounts to arguing that running two applications on the JVM makes them interoperable. Two EVM-based applications built on two EVM-compatible chains do not communicate or interact with each with much more ease than any two applications. Using standards like ERC-tokens and common tooling like web3.js *can* make building interoperability solutions easier as access patterns are uniform across both sides. But that can also be achieved using API-abstraction layers like Hyperledger Cacti[89].

The second sentence, however, does allude to a type of interoperability that EVM compatibility *can* enable if done right. "*applications that can be used across multiple blockchain networks"* describes exactly the idea of multi-venue assets and applications that Code Reusability enables. If the multi-venue EVM application is able to interact with *native* applications on all its venues using smart contract calls, then it can offer meaningful interoperability. For example, USDC[90] provides meaningful interoperability between the Ethereum and Solana ecosystems. USDC is a multi-venue asset that is natively interoperable with assets on both sides. A user can freely exchange their USDC on Ethereum for USDC on Solana through Circle. Thus any asset on Ethereum can be exchanged with low friction for any asset on Solana by going via USDC. This is not atomic and does involve an extra counterparty (Circle), but it's a sound alternative to going through an exchange and shows how multi-venue applications can provide interoperability.

Solidity and EVM are two sides of the same coin just like Daml and Canton are. The language market share statistics above show that Solidity is the only language with meaningful market share targeting the EVM, and Solidity has one primary compiler[91]. At the level of the diagram in section 2.5, the EVM stack then looks rather similar to the Daml stack.

---

[88] Solving interoperability in asset tokenisation, Adam Belding, Calastone, https://www.calastone.com/insights/unlocking-the-future-solving-interoperability-in-asset-tokenisation/
[89] Hyperledger Cacti, https://www.lfdecentralizedtrust.org/projects/cacti
[90] USDC website, https://www.circle.com/en/usdc
[91] Solidity github repo, https://github.com/ethereum/solidity



While the Solidity language at the core uses ECMAScript for its expressions, just like Daml uses Haskell, it adds many extensions designed to efficiently express the EVM ledger model, just like Daml has primitives for Canton's ledger model. The EVM ledger model is exposed through the Ethereum JSON RPC API, just like the Canton ledger model is exposed through Canton's Ledger API. Users and off-ledger app components interact with the ledger model through that API. Neither app, user, nor developer interact meaningfully with the actual virtual machine (EVM), nor the EVM bytecode (contracts).

Valuable EVM compatibility for Canton Network**,** in light of the above and the potential for EVM privacy on Canton discussed in section 1, therefore consists of:

Solidity support, meaning the ability for developers to reuse as much Solidity code as they can, with as few alterations as possible, and having it executed faithfully on Canton according to the EVM ledger model.

JSON RPC API support, meaning the off-ledger components can interact with Solidity contracts and the EVM ledger model running on Canton in a way that is as faithful as possible to Ethereum.

Native interoperability, meaning Solidity contracts running on Canton can make atomic smart contract calls to native Canton contracts and vice versa.

Controls, meaning the nodes that participate in the consensus on Solidity contracts running on Canton can be configured freely.

Privacy and Confidentiality, meaning the contract state of different Solidity contracts is distributed selectively, just like Canton contracts are distributed selectively. The observers of Solidity contract state can be configured freely.

Given the differences between the Canton and Ethereum ledger models and tech stack below the level presented above, there are numerous challenges to overcome, but with the potential to add privacy and control to EVM contracts, there are also interesting

opportunities beyond the adoption and code reusability of regular EVM compatibility. The rest of this section presents a high level plan for building up EVM compatibility in Canton.

Section 5.1 covers the work involved in laying the groundwork for running any Solidity on Canton, by making *pure* Solidity code reusable. It'll cover how even this already allows for significant code reuse for real world applications, with native interoperability.

Section 5.2 will cover the relatively short gap between running pure Solidity code, and Solidity contract support with native interoperability as described above, achieved through coarse ledger model mapping. It'll cover the limitations of this approach, but also show how this provides coarse but important privacy and confidentiality properties beyond any other EVM compatible ledgers.

Section 5.3 will talk about JSON RPC API support. This is a critical component in solving for code reusability, but technically it is the most mundane.

Section 5.4 expands on 5.2, describing how Ethereum's storage trie structure might be used to perform much more fine-grained ledger model mapping, lifting some of the limitations in 5.2, but also leading to fine-grained confidentiality for Solidity contracts. Such fine-grained mapping will likely need to be supported by developer annotations to specify visibility of different parts of Solidity contract state and events.

5.1 Manual EVM Orchestration

The foundation for EVM compatibility is the ability to execute Solidity code inside a virtual machine supported by Canton, and to call that code from choice bodies of Canton native contracts, in particular. This would allow Solidity code to be used as a library in Canton smart contracts written in another language. How much code reuse this would allow depends on the code base in question, but it may be significant. Take, for example, the ForgeBond contract[92] developed for the bond issuance by the SocGen Forge for the EIB bond issuance[93]. The code base consists of a single contract (ForgeBond). The contract has a state encoded in four structs:

```
BasicTokenLibrary.BasicToken private token;
BasicTokenLibrary.Bond private bond;
OperatorManagerLibrary.OperatorManager private operatorManager;
SettlementRepositoryLibrary.SettlementTransactionRepository
    private settlementTransactionRepository;
```

Beyond manipulating these structs, the only interaction with the blockchain is I/O via emitting events.

Basic Reuse of such a contract on Canton is to manually orchestrate an EVM to

1. Deploy the Solidity contract.
2. Manipulate and read from the Solidity contract by calling functions on it.
3. Read and process emitted events.

[92] ForgeBond source code via Blockscan, https://vscode.blockscan.com/ethereum/0x1Ff3D45E2c6c638A8d6BD1c81c99E6dB6D585EEb
[93] Digital Innovation In Capital Markets, Societe Generale, European Investment Bank, Forge, https://www.ecb.europa.eu/paym/groups/pdf/omg/2022/220922/Item_2%20_Digital_Innovation_in_Capital_Markets.en.pdf

This effectively already allows for manual implementations of the patterns proposed in 5.2 and 5.4 by adding a new constructor, and a new read function on the contract to extract state in bulk. For an app like the aforementioned EIB bond which has a single contract with state encapsulated in four structs, and with 125 transactions in its entire lifetime[94], manually implementing the pattern in 5.2 is not a hard problem, presents no scalability issues, and would allow for a lift-and-shift of the smart contract code to Canton with little overhead.

Calling the EVM from within a Rust-written Wasm contract for Canton might follow a similar pattern to ethcontract-rs[95]. Here an illustrative example how a contract might be instantiated and manipulated:

```
// Illustrative EVM interaction from Canton
// Inspired by ethcontract-rs
use web3::types::*;
use canton::evm::transaction::Address;

// macro to generate a `MyContract` type consisting of bytecode,
// linking to an isolated evm instance, and type-safe bindings to
// contract functions.
// Take some ERC20 sample as an example.
canton::evm::contract!("path/to/MyERC20.json");

// ...

//In the context of a choice:

// Artificial addresses, which may be read from the contract state.
let owner: Address = "0x0".parse()?;
let receiver : Address = "0x1".parse()?;

// now create an instance of a solidity contract
// assuming an empty constructor.
let erc20 = MyERC20::deploy().from(owner).execute();

// Call a mutating function
erc20.transfer(receiver, 1_000_000.into()).from(owner).execute();

// Call a view function
erc20.balanceOf(receiver).from(owner).execute();

// Get events
let mut transfers = erc20
```

[94] ForgeBond on Etherscan, https://etherscan.io/address/0x1Ff3D45E2c6c638A8d6BD1c81c99E6dB6D585EEb
[95] ethcontract-rs GitHub repo, https://github.com/cowprotocol/ethcontract-rs/tree/main

```
.events()
.transfer()
.from(Topic::This(owner))
.execute();
```

Hosting Solidity, as illustrated by section 4, can be done in one of two ways. Either one can compile Solidity to an existing smart contract engine, or one can embed a new smart contract engine that supports the language in question. Concretely, assuming Wasm support is present, this means either hosting an EVM implementation in Canton participants as a separate VM, compiling Solidity to Daml-LF, compiling Solidity to Wasm, or running an EVM inside the Daml or Wasm engines. There is no prior art for the Daml options so they are unlikely paths. Canton is JVM-based, and community support for JVM-hosted EVMs does exist, so hosting the EVM as a separate virtual machine next to Daml and Wasm is one option. Building on Wasm also seems possible thanks to well maintained open source projects, both already in use in the Polkadot ecosystem. This presents a total of three possible options:

Cross-compilation via Hyperledger Solang[96], an open source Solidity compiler targeting non-EVM bytecode, is one possible route. Solang has an existing compiler backend targeting Wasm bytecode built to target Polkadot.

EVM-hosting inside Wasm using Rust EVM[97] (also known as SputnikVM) is another route. It claims to be hostable in WebAssembly (Wasm), and forms the core of the Polkadot Frontier EVM compatibility layer[98].

EVM-hosting as a separate VM inside the JVM, for example based on Hyperledger Besu[99].

Either of these approaches is likely to go a long way towards being able to instantiate and read Solidity types in a Wasm-hosted smart contract language, and to deploy and call Solidity contracts from that host.

EVM host function implementation will be the biggest challenge beyond basic integration, independent of which approach is taken. The host functions of the Rust EVM[100] as well as their implementation in Frontier's EVM stack-based runner[101] illustrate the complexity of doing so. For the purpose of the manually orchestrated EVM, the storage backend can likely be kept to a simple in-memory store. The host functions not related to storage are largely context related and exposed in Solidity via block and

[96] Hyperledger Solang, GitHub repo: https://github.com/hyperledger/solang, Docs: https://solang.readthedocs.io/
[97] Rust EVM GitHub repo, https://github.com/rust-ethereum/evm
[98] Frontier documentation, https://github.com/polkadot-evm/frontier/blob/master/frame/evm/README.md#evm-engine
[99] Hyperledger Besu, GitHub repo: https://github.com/hyperledger/besu/
[100] Rust EVM Runtime Backend Traits on GitHub, https://github.com/rust-ethereum/evm/blob/master/interpreter/src/runtime.rs
[101] Frontier implementation of Rust EVM backend on GitHub, https://github.com/polkadot-evm/frontier/blob/master/frame/evm/src/runner/stack.rs

transaction properties[102]. The majority can be zeroed out or not supported for EVM support in Canton:

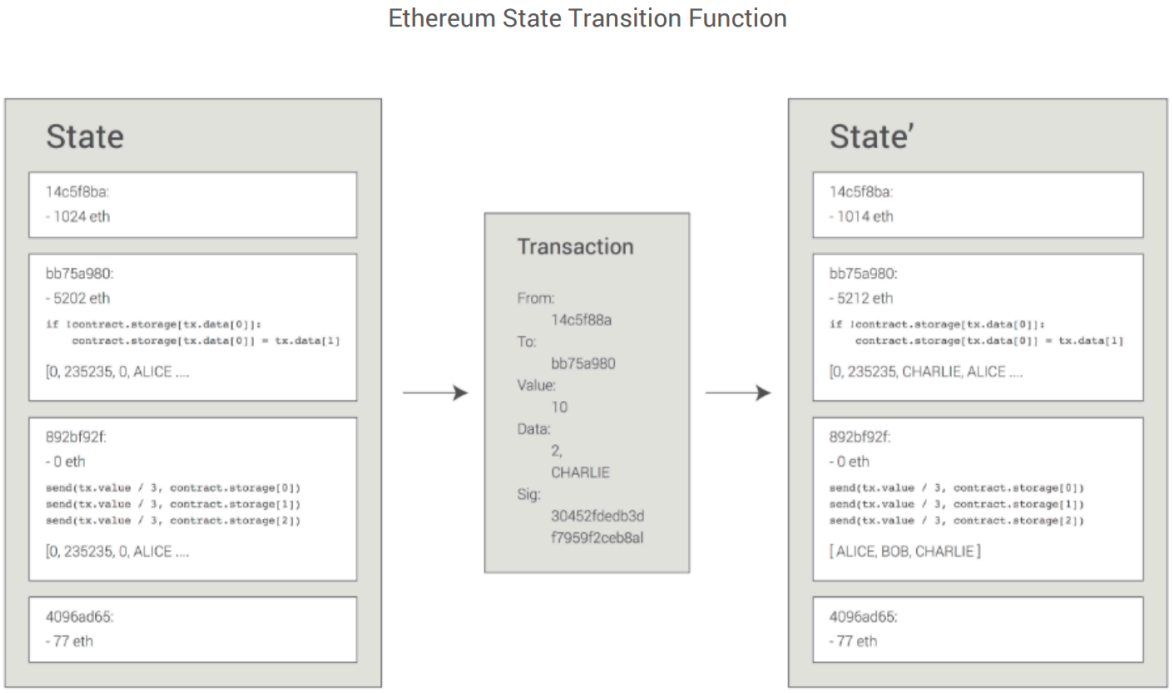
Property / Function	**Canton Mapping**
blockhash(uint blockNumber) returns (bytes32) block.number (uint)	Not supported. Always returns 0. See 5.4 for reasons.
blobhash(uint index) returns (bytes32) msg.data (bytes calldata) msg.sig (bytes4)	No special treatment needed.
block.basefee (uint) block.blobbasefee (uint) block.difficulty (uint) msg.value (uint) tx.gasprice (uint)	Always returns 0.
block.chainid (uint)	Always 0 in library mode. Hash of the EVMInstance contract key in section 5.2 and beyond.
block.coinbase (address payable)	Equals tx.origin.
block.gaslimit (uint) gasleft() returns (uint256)	Maximum integer value.
block.prevrandao (uint)	Not supported.
block.timestamp (uint)	Canton Ledger Effective Time
msg.sender (address) tx.origin (address)	Always 0 in library mode. Starts with calling party address in 5.2 and beyond.

Manual EVM orchestration allows for broad code reuse of Solidity contracts in Canton. But it is left to the developer to extend their Solidity code with appropriate state im- and export functions, and to map storage and events to Canton native constructs. To get to true lift and shift code reusability as well as seamless interoperability between Solidity and Canton Native contracts, additional functionality and pre-packaged patterns are needed. These are demonstrated in sections 5.2 and 5.4.

[102] Solidity Block and Transaction properties, https://docs.soliditylang.org/en/develop/units-and-global-variables.html#block-and-transaction-properties

5.2 Coarse Ledger Model Mapping

The EVM is a state machine. It has an internal state S, and every function either reads from S via function r(S), or it writes to S, mutating the state to some function of S, S' = w(S).



Source: Ethereum Whitepaper

This makes it relatively straightforward to use the capabilities from section 5.1 to functionally map the EVM ledger model to the Canton ledger model, or indeed any UTXO model. A single UTXO which keeps the entire EVM state S and gets consumed and re-created with data S'=w(S) by any function call to w behaves exactly like the state machine.

If it were possible to read and load EVM state wholesale from a Canton serializable value, one could write a Canton contract representing an EVM instance as follows. While this is likely to happen from within a Wasm-hosted language, the below uses Daml syntax for its conciseness.

```
template EVMInstance
 with
   sigs : [Party]
   obs : [Party]
   state : EVMState
 where
   signatory sigs
   observer obs
```

```
nonconsuming choice Call : (ContractId EVMInstance, [EVMEvent])
   with
     signedTx : EVMSignedTX
     caller : Party
   controller caller
   do
     evm <- initializeEVMFromState state
     h <- stateRootHash evm
     events <- submitSignedTx evm signedTx
     h' <- stateRootHash evm

    forA_ events (\event -> exercise self Emit with event)

     if h == h'
       then return (self, events)
       else do
         archive self
         state' <- evmState evm
         self' <- create this with state = state'
         return (self', events)

nonconsuming choice Emit : ()
   with
     event : EVMEvent
   observer obs
   controller sigs
   do ()
```

The function submitSignedTx is already enabled by 5.1. The serializable EVMState and EVMSignedTx types as well as the functions initializeEVMFromState, evmState, and stateRootHash are relatively minor extensions to the work in 5.1. The above template could be made available as a library to enable Solidity contracts on Canton without any custom development.

EVM Events in the above are made available via the Canton Ledger API as non-consuming choices on the EVM instance.

One-way Interoperability between Canton native contracts and EVM is already given by the above. Canton contracts can call into EVM instances. This enables constellations of privacy and atomicity beyond any other private Ethereum implementation today.

Returning to the DvP example from earlier, we imagine Issuer1 has developed a payment system, and Issuer2 has developed a bond token. Each token should be visible to and validated by small, nonidentical groups of entities. One entity, called Exchange in

this example, with visibility into both tokens, offers swap/exchange services. Alice and Bob have entered into a trade.

With the above EVMInstances on Canton, this is easy. Issuer1 and Issuer2 each deploy an EVM instance with them as the respective signatory and the Exchange as a common observer. Alice and Bob independently allocate funds to the trade by delegating them to the Exchange. The Exchange can send two Canton commands, or call the two EVM instances from a single choice and thus make the two transfer calls in a single atomic transaction. Issuer1 and Issuer2 see only the transfers on their respective EVM instances.

Contrast this with today's capabilities of subnets, rollups, groups, channels, or private Ethereum instances. The two transfers have to happen in two independent transactions. De facto, the exchange has to act as a central counterparty, facilitating traditional non-atomic clearing and settlement. The exchange can reduce the risk of partial settlement by implementing custom protocols like hashed timelock contracts[103], but they cannot eliminate it.

Authorization in the above is solved crudely in that it does not link Canton's authorization model with the EVM's. The addresses in the EVM are EVM externally owned addresses (EOA), and a signed Ethereum transaction has to be passed through a Canton command. As a result, in the above example, the exchange is the only guarantor of atomicity of the swap. They could choose to send a single transfer command rather than both. It would be nicer if the trading parties, which may be financial institutions themselves, could participate in consensus and thus guarantee atomicity for themselves. In other words, it would be preferable if the trading parties themselves would have to authorize the call to the EVM instead of delegating the funds to the exchange.

This can be accomplished with identity mapping between parties and addresses, not requiring a signature on the EVM translation, and instead setting the EVM's tx.origin (and initial msg.sender) to be the controller party of the Call choice. One could call such addresses *Canton Party addresses* to distinguish them from EOAs.

```
            events <- submitTx evm caller tx
```

In the EVM, an address is a 160-bit fingerprint of a public key. It would be natural to translate a Canton Party to an EVM Address by taking a 160-bit hash of the full party identifier. With this capability in place, the Swap could be conducted as if it were a Swap between Canton native tokens. Note that in this scenario, the fromParty and toParty will see the EVM instances being manipulated, so they learn of the token state at the point of the swap. Section 5.4 will address this topic.

```
      template EVMSwap
       with
         fromParty : Party
         toParty : Party
```

[103] Lightning Networks Part II: Hashed Timelock Contracts (HTLCs), Rusty Russell, 2015, https://rusty.ozlabs.org/2015/04/01/lightning-networks-part-ii-hashed-timelock-contracts-htlcs.html

```
        exchangeParty : Party
        fromToken : EVMAddress
        toToken : EVMAddress
        fromAmount : Decimal
        toAmount : Decimal
      where
        signatory fromParty, toParty, exchangeParty

        choice Swap : ()
          with
            fromEVM : ContractId EVMInstance
            toEVM : ContractId EVMInstance
          controller exchangeParty
          do
            exercise fromEVM Call with
                tx = toERC20Transfer (fromToken, fromAmount, toParty)
                caller = fromParty
            exercise toEVM Call with
                tx = toERC20Transfer (toAddress, toAmount, fromParty)
                caller = toParty
```

Full Interoperability requires the ability to call from EVM instances to Canton native contracts, and equivalently from EVM instance to EVM instance. The above already hints at one of the challenges to be overcome, which is addressing. The EVMAddress types above refer to smart contracts within the EVM instances. In a single EVM, addresses are unique. With the potential for multiple EVM instances, we have to make sure they are also unique in Canton and can be resolved to the right EVM instance.

Smart Contract Addresses in the EVM are also 160-bit hashes as introduced for Party mapping above. They are a hash of the sender's address plus the sender's nonce used for the contract deployment. A similar scheme might be possible for Canton, but has the challenge that to avoid the same address existing in two EVM instances, each Party would need to maintain a global nonce. A likely better alternative is to calculate smart contract addresses not only from sender and nonce, but from the triple (sender, sender nonce in EVM instance, EVM instance Id).

This leaves open how EVM instances are identified. A natural choice would be to use Daml's contract keys, and rely on the signatories of an EVM instance to ensure uniqueness of the identifier. This context could be passed into every EVM call so that addresses could be computed appropriately.

```
template EVMInstance
 with
```

```
 uuid : Text
  sigs : [Party]
  obs : [Party]
  state : EVMState
where
  signatory sigs
  observer obs
  key (uuid, sigs) : (Text, [Party])
  maintainer key._2

  nonconsuming choice Call : (ContractId EVMInstance, [EVMEvent])
    with
      callData : EVMCallData
      caller : Party
    controller caller
    do
      evm <- initializeEVMFromState state (key this)
```

EVM instance resolution still requires some sort of persistent index, which would need to be provided by the hosting participant, further simplifying the Swap choice.

```
  choice Swap : ()
    controller exchangeParty
    do
      callEVMAddress fromToken with
          tx = toERC20Transfer (fromAmount, toParty)
          caller = fromParty
      callEVMAddress toToken with
          tx = toERC20Transfer (toAmount, fromParty)
          caller = toParty
```

Note that ignoring the explicit caller party, this is now equivalent to a contract call within Solidity itself.

```
fromToken.transfer(toAddress, 1000);
```

The Canton-hosted EVM (or Wasm code) can thus make a dynamic choice. If the fromToken address is known in the current EVM instance, call it EVM-instance internally. If it is not, insert an exercise node corresponding to a callEVMAddress fromToken with the tx.origin being carried over, and msg.sender set appropriately.

EVM-to-Canton calls would require an always-available pseudo-contract in each EVM-instance, likely at a fixed address. Calls to this pseudo-contract would need to be

intercepted by the host and treated appropriately. Such a call would have the authority of the party used in tx.origin.

```
pragma solidity ^0.8.3;

// Canton library and interface provided out of the box
library Canton {
   address constant canton = 0x0000000000000000000000000000000000000001;
   ICanton constant interfaceContract = ICanton(canton);

   struct CantonCall {
       string cid;
       string choice;
       bytes args;
   }

   function call(CantonCall memory params) returns (bytes memory ret) {
       interfaceContract.callCanton(params);
   }
}

interface ICanton{
   function callCanton(Canton.CantonCall memory params)
       returns (bytes memory ret) ;
}


// User code
contract Caller {
 constructor () public {}

 function interactWithCanton () {
   bytes ret = Canton.call(CantonCall(foo, bar, baz))
 }
}
```

Code Generation could be used either side to make EVM - Canton and Canton - EVM calls more type safe. The way this might work to call an EVM contract from Canton was already demonstrated using Rust syntax in 5.1. Vice versa, creating appropriate EVM structs for data types of Canton-native template and choice arguments and return types would make it easier to handle calls than needing to en-/decode bytes.

The above construct is highly powerful in that it allows EVM execution in Canton with privacy, and enables two-way interoperability between Canton native and EVM contracts. However, it has three limitations.

Contention is a problem account based ledger models like EVM and UTXO based ledger models like Canton solve differently. In UTXO models, commands are interpreted before ordering. Contention is detected through collision detection on UTXOs and attempted double spends are rejected. To get low contention, developers need to break up the state into appropriately small UTXOs. Account based models like EVM interpret commands after ordering. This has downsides like reordering and frontrunning attacks, and harder parallelizability, but it also has the upside that transactions don't fail due to simple write-write contention or even read-write contention. Translating the ledger model to UTXO as proposed here means that every two writes to a single EVM instance contend with each other. As a consequence, throughput on a single EVM instance is limited. Going back to the throughput requirements of a token like the EIB bond, this is unlikely to be a practical issue for initial lift-and-shift uses. Section 5.4 will address this limitation.

Reentrancy[104] is both a major feature of the EVM as well as one of its bigger security flaws. UTXO based ledgers like Canton do not allow for re-entrancy. Since a single EVM instance in the above is archived and re-created exactly once for a call, re-entrancy within a single EVM instance works just fine. However, re-entrancy between two EVM instances will fail. Say there are EVMInstance contracts A and B. A function foo within A calls to a function bar within B, which in turn calls a function baz within A. Both the calls to foo and baz will attempt to archive A through the archive self call inside the Call choice. This is a double spend in the UTXO model and will therefore not work.

This is a form of *contention* as the original call to A, and the re-entrancy from bar are treated as two separate calls, and thus run into write-write contention. As such, 5.4 will also address *some* reentrancy limitations.

Privacy and Confidentiality in this model is already superior to "private EVM" ledgers in that atomic transactions across different stakeholder sets are possible while maintaining privacy as seen in the DvP example.

However, the DvP example also shows that the privacy achieved here is inferior to Canton's native capabilities. With Canton's native sub-transaction privacy Alice and Bob can participate in consensus on the Swap while only learning about the details of the Swap that they know anyway. In the coarse ledger mapping presented here, Alice and Bob learn the entire EVMInstance state if they participate in consensus.

Scalability could become an issue for large Solidity contracts. Canton is optimized for small data payloads on contracts. While it has been proven to work with single contracts storing upwards of 100MB, most synchronizers limits message sizes to just 10MB, which includes both input and output contracts. So in practice, to perform a DvP, the total data on each of the two involved EVM instances will need to be below 2MB. Assuming an optimal 64 bytes per balance entry (256 bit each for slot hash and value), a simple ERC20 token would hit this limit at around 16384 balance entries. That's a non-trivial

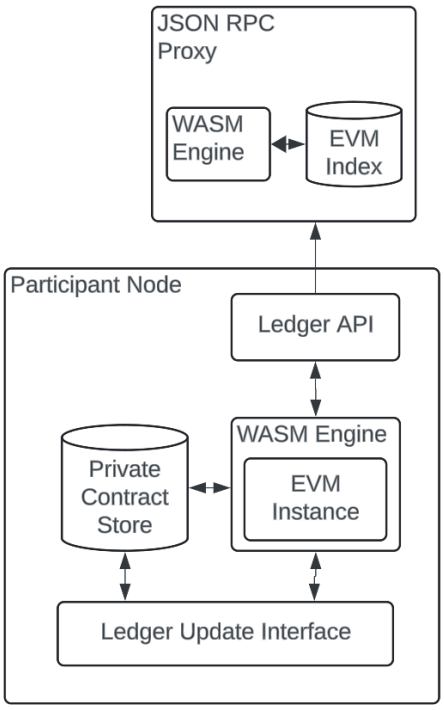
[104] Re-Entrancy, Solidity by Example, https://solidity-by-example.org/hacks/re-entrancy/

number and sufficient for many of the non-stablecoin Real World Assets (RWAs) today, but doesn't even come close to supporting a retail stablecoin like USDC.

Summarizing this section, building on the basic ability to instantiate and call EVM instances from Canton native contracts, it is possible to automatically generate contract and identity mappings that allow for functional lift-and-shift of EVM contracts to Canton. Canton's smart contract interoperability and privacy capabilities are transferable to these EVM instances allowing for atomic transactions spanning multiple EVM instances and Canton native contracts with different validator and observer sets. While some limitations will require developers to decide carefully how to partition their EVM code into EVM instances on Canton, and how to construct settlements to maintain privacy, this construct offers capabilities beyond any other EVM ledger, as well as EVM-compatibility and interoperability of Canton at a high fidelity.

5.3 API Wrappers

Off-ledger integration with Solidity contracts happens via the Ethereum JSON RPC[105]. To fully address the developer adoption and code reusability goals stated in the introduction of section 5, Canton must expose the Ethereum JSON RPC to interact with EVM instances running on Canton. The most likely approach for doing so is a proxy pattern similar to the Neon EVM targeting Solana[106].



For Canton, this involves putting an API proxy in front of Canton's Ledger API. The proxy consumes data from the Ledger API, possibly stores and indexes it in a form of persistent cache, and exposes read methods to clients. Vice versa, it takes calls to write methods and converts them to Ledger API Commands.

Since some Ethereum RPC methods require running smart contracts (e.g. eth_call), such a proxy also needs to run the smart contract engine that handles EVM contracts. Since the Ethereum JSON RPC is designed for a single EVM instance, a separate endpoint would be provided per EVM instance on canton, for example using the chain id (hash of EVMInstance contract key) as URL path.

The most important RPC methods for integration are the gossip, state, and history APIs that respectively allow for writing to an EVM chain, reading current state, and navigating historic state. The UTXO mapping from section 5.2 ensures that historic EVM Instance

[105] Ethereum JSON RPC docs, https://ethereum.org/en/developers/docs/apis/json-rpc/
[106] Neon EVM docs, How it works, https://neonevm.org/docs/about/how_it_works

state is available via Canton's Ledger API. The Canton <> EVM interoperability and addressing ensure that EVM transactions can be submitted through the Ledger API.

Blocks don't exist in Canton as they do in Ethereum, and blocks feature heavily on all three types of API methods. However, transactions on a single contract like an EVMInstance are consistently totally ordered for all signatories, so for API purposes, one can treat transactions as blocks and convert transaction ids into block ids through appropriate hashing: Ethereum Block Hash = H256(canton transaction id).

While block numbers are not available in Solidity (see section 5.1) to avoid contention in preparation for section 5.4, the JSON RPC Proxy could assign sequential block numbers to transactions affecting an EVM instance and thus also serve the RPCs that are ByNumber, and not ByHash.

Canton's consensus has instant deterministic finality so block default parameters map to either the earliest available EVMInstance state, latest committed EVMInstance state, or the given block number.

The lack of blocks also implies the lack of uncles[107]. All RPCs dealing with uncles would behave as if there are no uncles.

EVM transactions do not correspond one to one to Canton transactions. A single Canton transaction may contain multiple EVM transactions on a single EVM instance, as well as EVM transactions on multiple EVM instances. Due to the lack of re-entrancy across EVM instances, EVM transactions do, however, correspond one to one to Call exercises on EVMInstance contracts. Thus the RPC Proxy can appropriately hash, and index those events to serve EVM transactions both by Hash and Number, and serve EVM transaction receipts.

Block data doesn't exist as such. Thus when requesting a block by hash or number. The RPC would populate data like this:

number: QUANTITY	See above.
hash: DATA, 32 Bytes	Canton transaction id hash.
parentHash: DATA, 32 Bytes	Canton transaction id hash of the previous transaction that changed the same EVM instance.
nonce: DATA sha3Uncles: DATA, 32 Bytes logsBloom: DATA, 256 Bytes transactionsRoot: DATA, 32 Bytes stateRoot: DATA, 32 Bytes receiptsRoot: DATA, 32 Bytes	null/0

[107] What are uncle blocks, Alchemy documentation, https://docs.alchemy.com/docs/what-are-uncle-blocks

miner: DATA, 20 Bytes difficulty: QUANTITY totalDifficulty: QUANTITY extraData: DATA size: QUANTITY gasLimit: QUANTITY gasUsed: QUANTITY	
timestamp: QUANTITY	Canton Ledger Effective Time
transactions: Array	Implemented faithfully.
uncles: Array	[]

Transaction signing and its interplay with offchain components is core to the way Ethereum is integrated in user experiences and security solutions. The identity mapping in 5.2 means that Canton Party addresses are no longer public key fingerprints. Furthermore, the Canton Ledger API expects a signed Canton transaction or JWT-based authentication against a participant that hosts the submitting party in submission mode, so a signed Ethereum transaction does not help to authenticate against the Canton Ledger API.

To support Ethereum externally owned addresses (EOAs) securely, a unique party would have to be generated from the key corresponding to the EOA. This could be done by taking the EOA private key as a Canton namespace root key, and reserving some special prefix to designate the party corresponding to the EVM address. Suppose a user has generated private key P corresponding to address A. The user could now onboard their wallet to Canton by creating a namespace N from private key P's public key fingerprint, and allocate a party ("eth_address", N), where "eth_address" is a hard-coded reserved party name. They would furthermore have to write a delegation contract to the ledger, which allows the proxy provider to submit signed EVM transactions with that party/address combination. This allows the proxy provider to authenticate against the Canton Ledger API using their own party (proxy_operator below).

```
template EVMProxyDelegation
 with
   eth_address_party : Party
   proxy_operator : Party
 where
   signatory eth_address_party
   observer proxy_operator

   nonconsuming choice DelegatedCall : ()
     with
       evm : ContractId EVMInstance
       callData : EVMCallData
     controller proxy_operator
```

```
    do
      validateEthSignatureForParty eth_address_party callData
      exercise evm Call with
        callData = callData
        caller = eth_address_party
```

Such a construct would allow Ethereum wallet clients like MetaMask to work with Canton hosted EVMs and even interact with Canton native contracts through on-ledger interoperability. The key owner could set up the same key for external signing of Canton native transactions, providing a *unified wallet* across EVM and canton native assets.

Summarizing, a JSON RPC proxy constructed like this would be able to serve the core gossip, state, and history RPC methods to a degree of compatibility that supports lift and shift of Ethereum application to Canton with only minor caveats or modification.

Users could interact with a single unified ledger through Canton or EVM native interaction patterns or APIs, up to and including using the same signing key Canton and EVM native interactions offering something akin to a unified wallet across tokens native to the different ledger models.

5.4 Fine grained mapping with added privacy

The limitations in section 5.2 are all a result of the coarse state mapping of one EVM instance to one Canton contract. Such monolithic UTXOs go against the general ethos and design of eUTXO systems. To solve the contention issue, in particular, data has to be structured into Canton contracts in such a way that transactions by two users do not attempt to consume any common contract. A simple ERC-20[108] token serves as an example.

```
contract ERC20 is IERC20 {
   event Transfer(address indexed from, address indexed to, uint256 value);
   event Approval(
       address indexed owner, address indexed spender, uint256 value
   );

   uint256 public totalSupply;
   mapping(address => uint256) public balanceOf;
   mapping(address => mapping(address => uint256)) public allowance;
   string public name;
   string public symbol;
   uint8 public decimals;

  constructor(string memory _name, string memory _symbol, uint8 _decimals) {
```

[108] Solidity By Example, ERC-20, https://solidity-by-example.org/app/erc20/

```
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
    }

    function transfer(address recipient, uint256 amount)
        external
        returns (bool)
    {
        balanceOf[msg.sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }
```

Contention free transfers would require the state corresponding to each map entry in balanceOf to be stored in a separate Canton contract. The only contract state mutated by transfer are the balanceOf entries corresponding to message sender and transfer recipient. If address 0x01 transferred funds to address 0x02, and address 0x03 transferred funds to 0x04, then those four pieces of state would reside on separate Canton contracts. The two transactions would consume disjoint Canton contracts and not contend on the balances.

To extend this to the transaction as a whole, the transaction must not consume any other shared Canton contracts. In particular, the "global" state of the ERC-20, meaning totalSupply, name, symbol, decimals must not be consumed, nor must any other map entries. And there must not be any mutable shared state on the EVMInstance itself. This is the primary reason that the EVM in Canton would not make available block numbers or block hash. Nor can *World State* or *Account State* hashes for contract address be stored on the blockchain. More on those below.

High Scalability would also be achieved by storing the balances of different asset holders in separate UTXOs. The balanceOf (and allowance) maps represent the vast majority of the state of a typical token. If this state is broken up across many Canton contracts, only a few of which are used as inputs to a transfer, then individual transactions and contracts can stay small in size and there is no limit to the number of holders or total transfer throughput on a given token.

The EVM's storage model and typical Solidity contracts like the above or the ForgeBond example are amenable to being mapped to a UTXO ledger in such a way that frequently mutated state associated with a single address is stored in separate UTXOs. For a full explanation of Ethereum's storage architecture, the reader should refer to the Ethereum Yellow Paper[109]. What's important to understand for this paper is that Ethereum's storage

[109] Ethereum Yellow Paper, https://ethereum.github.io/yellowpaper/paper.pdf

consists of two nested levels of Merkle Patricia Tries[110]. The outer tree called the *World State* maps 160 bit addresses to *Account State*. Account states have four fields:

- *nonce*: A counter that is incremented each time the address submits a transaction. For contract accounts, the nonce is incremented only on the CREATE operation, meaning only if another contract is created, not if another contract is called[111].
- *balance*: The account's balance of the native currency (e.g. ETH).
- *storageRoot*: A reference to the storage trie of the account. This is empty for externally owned accounts.
- *codeHash*: A reference to the code deployed to the account. This is empty for externally owned accounts.

The storage root of an account is the root hash of another Merkle Patricia Trie mapping 256 bit keys to 256 bit values. Each entry is called a *slot*. The solidity compiler assigns each *storage* variable on a contract a slot[112]. It does so in order, meaning the first variable goes into slot 1, and it uses packing, meaning data types shorter than 256 bit may share a slot. For variable length types like arrays, bytes, string, or maps, the variable itself takes up exactly one slot, but its values are mapped to other slots by hashing their keys and relying on no collisions in the 256-bit key space of the trie. Importantly here, for a mapping like balanceOf in slot p, say, the value corresponding to key k would be found in slot keccak256(h(k) . p), where the dot means concatenation. This extends to nested maps like allowance at slot q, say. The value of entry allowance[k1] would be at s(k1) = keccak256(h(k1) . q). The value of allowance[k1][k2] would be at keccak256(h(k2) . s(k1)).

Mapping the Slots to UTXOs gives exactly the contention properties needed for contention free transfers. A rough schema using Daml syntax is shown below.

```
template EVMInstance
 with
   uuid : Text
   sigs : [Party]
   obs : [Party]
 where
   signatory sigs
   observer obs
   key (uuid, sigs) : (Text, [Party])
   maintainer key._2

   nonconsuming choice Call : ()
   with
```

[110] Merkle Patricia Tries, https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/
[111] EIP-161 specifying contract account nonce behaviour, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-161.md
[112] Solidity Storage Layout, https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html

```
    callData : EVMCallData
      caller : Party
    controller caller
    do
      events <- callEVM evm callData
      forA_ events (\event -> exercise self Emit with event)

template AccountState
 with
   evm : EVMInstance
   nonce : Int
   -- No Balance as always 0.
   address: Address
   externalParty : Optional Party
   code : Optional EVMCode
 where
   -- Inherit metadata from the evm Instance
   -- Except that parties can self-sign their own
   -- AccountState for externally owned accounts
   signatory (signatory evm) , externalParty
   observer (observer evm)
   key (key evm, address) : (EVMInstance, Address)
   maintainer (maintainer key._1)

template Slot
 with
   accountKey : (EVMInstance, Address)
   slotNumber : Int
   slotValue : Int
 where
 -- Same metadata from the AccountState
 signatory (signatory accountKey._1)
 observer (observer accountKey._1)
 key (accountKey, slotNumber) : ((EVMInstance, Address), Int)
 maintainer (key._1._1)
```

The callEVM function needs to do more heavy lifting than in the section 5.2 model. Instead of being instantiated from a given state, running in memory, and then writing state back, it now needs to fetch, archive, and create state on the fly. All the read operations in Rust-EVMs RuntimeBaseBackend[113] would translate to Daml's fetchByKey operations. Write operations like set_storage in Rust-EVMs backend would translate to pairs of archive and create operations.

[113] Rust EVM RuntimeBaseBackend, https://github.com/rust-ethereum/evm/blob/master/interpreter/src/runtime.rs#L110

Non-existence checks of contract keys are not supported natively by Canton's ledger model whereas non-existence checks of map keys are commonplace. To resolve this, a low-contention Radix-tree index could be maintained on-ledger for each map in Ethereum.

The above mapping of the Ethereum account storage trie to Canton contracts resolves the contention, re-entrancy, and scalability limitations of 5.2. It can also address confidentiality. The EVM instance operators would need to create read access controls at the API layer so that Alice can read exactly the state needed for her to submit transfers, and Alice would need to supply them to her participant node (or Proxy provider) using Canton's explicit disclosure feature[114]. This is workable, but not as nice as using the on-ledger observer feature. Furthermore, the above is rather expensive on the runtime. The global state variables, totalSupply, name, symbol, and decimals, each reside in their own Canton contract, but are likely accessed for many transactions. This is pronounced even more in the real-world ForgeBond example where there is more global state on the contract.

Custom annotations to Solidity could let the developer choose which pieces of state are stored in their own, separate Canton contracts, and dynamically add observers to fully utilize Canton's privacy features for EVM state sharing. The @ annotations in the below are illustrative only. Depending on the implementation approach (Wasm hosted EVM vs compilation to Wasm) a different type of annotation would work better. For example implementing a provided interface with pure functions mapping events and key/value pairs to observers.

```
contract ERC20 is IERC20 {
   @observers([e.from, e.to])
   event Transfer(address indexed from, address indexed to, uint256 value);

   @observers([e.owner, e.spender])
   event Approval(
       address indexed owner, address indexed spender, uint256 value
   );

   uint256 public totalSupply;
   @contractPerEntry
   @keyAsExtraObserver
   mapping(address => uint256) public balanceOf;
   @contractPerEntry
   @keyAsExtraObserver
   mapping(address => mapping(address => uint256)) public allowance;
   string public name;
   string public symbol;
```

[114] Explicit Contract Disclosure, https://docs.daml.com/app-dev/explicit-contract-disclosure.html

```
    uint8 public decimals;
```

Such a construct for native maps will likely need significant modification to a Solidity compiler and/or Runtime. The reason is that EVM bytecode has no notion of mappings, but uses the above rules for mapping map keys to storage slots. And there is no way to recover the storage key, which is the desired observer, from the storage slot. If this problem is intractable for native maps, developers may have to adjust their code from native mappings to custom mapping so that the host can do reverse lookups from slots to addresses. This could be accomplished, for example, by emitting a specific event rather than writing to storage directly, which can be intercepted and interpreted by the host.

```
library CantonMapping {
    struct Map {
        mapping(address => uint256) values;
    }

    event MapStore(uint256 mapSlot, address key, uint256 value);
    event MapDelete(uint256 mapSlot, address key);

    function get(Map storage map, address key) public view returns (uint256) {
        return map.values[key];
    }

    function set(Map storage map, address key, uint256 val) public {
      // Emit an event instead of setting storage directly.
      // The host is responsible for updating storage by creating
      // or cycling the Canton contract for the slot.
      uint256 mapslot;
      assembly {
          mapslot := map.values.slot
      }
      emit MapStore(mapSlot, key, val);

      // Compatibility for other EVM ledgers.
      if(map.values[key] != val) {
        map.values[key] = val;
      }
    }

    function remove(Map storage map, address key) public {
     uint256 mapslot;
      assembly {
          mapslot := map.values.slot
```

```
    }
    emit MapDelete(mapSlot, key);
    delete map.values[key];
  }
}
```

Any solution to such annotations will be costly to implement in Canton and maybe impose some work on the developer as in the above. But this cost is justified. If developers could annotate events and maps with observer information, a Solidity token could make use of Canton's full confidentiality capability while maintaining Solidity's programming model, and without needing to resort to API-level read permissioning.

6 Conclusion

Blockchain and distributed ledger technology have the potential to fundamentally transform the plumbing of the financial system, providing lower risk, more real time experiences and more integrated capabilities up to and including straight-through processing at the industry level. New value and business models are possible on smart contract blockchain platforms based on the ability to perform low trust atomic transactions across independent applications. Canton is a next generation layer 1 protocol providing the configurable controls, privacy and confidentiality allowing a wide spectrum of use cases and users, up to and including regulated financial institutions, to participate in the public permissioned Canton Network and extract the full benefits of the technology.

Daml is Canton's original smart contract language, designed from the ground up to safely and concisely program Canton's ledger model. It will remain a strong choice for programming Canton applications, but advances in virtual machines, and developments in the public permissionless blockchain space have opened the door to integrating additional smart contract programming languages for Canton. This paper presents two additive efforts to open up Canton to different languages and add value to the blockchain and DLT space as a whole by doing so.

Integration of a Wasm virtual machine as an alternative to the Daml engine would allow Canton native smart contracts to be programmed in a general purpose surface language like Rust or AssemblyScript. This would allow easier developer adoption of Canton by providing a more familiar experience than the purely functional and strongly typed Daml language. This work could well open the door to Canton being able to support an open ecosystem of virtual machines, opening up the Canton smart contract language ecosystem entirely.

Based on the Wasm virtual machine, or by integrating a separate virtual machine, it is possible to support Solidity smart contracts on Canton offering high fidelity EVM-compatibility. This would allow for the lift and shift of existing Solidity-based solutions to Canton and also open up the Canton Network to the existing Solidity developer community. But perhaps most importantly, EVM support as presented in this paper would bring Canton's configurable controls, privacy, and confidentiality to EVM contracts. This is

a capability no other chain or network can currently offer and presents a major hurdle to institutions moving beyond private permissioned deployments of EVM chains.

The materials presented in this paper are the first steps towards enabling the Canton Network ecosystem, Digital Asset included, to develop additional language support for Canton as open source contributions.