Canton: A Daml based ledger interoperability protocol*

Digital Asset Canton Team
canton@digitalasset.com
http://canton.io

2020-02-04

Abstract

Building distributed applications that involve multiple organizations is hard with today's technology. For each application, all organizations must agree on the data encoding, transport mechanisms, and interaction rules, and all must implement their part of the interaction correctly, including authentication and authorization. Once implemented, such an application often becomes a silo: there is no general convenient, secure and privacy-preserving way to integrate such applications and compose the business workflows that they automate. Smart contract platforms such as Ethereum try to break these silos, but suffer from scalability, authorization and privacy problems. Daml is a smart contract programming language whose distinguishing features are the built-in models of authorization and privacy. Canton is a next-generation distributed Daml runtime that implements these models faithfully. By partitioning the global state it solves both the privacy problems and the scaling bottlenecks of platforms such as Ethereum. It allows developers to balance auditability requirements with the right to forget, making it well-suited for building GDPR-compliant systems. Canton handles authentication and data transport through our so-called synchronization domains. Domains can be implemented in different ways depending on trust requirements. Domains can be deployed at will to address scalability, operational or trust concerns. They are permissioned but can be federated at no interoperability cost, yielding a virtual global ledger that enables truly global composition of business workflows.

1 Introduction

Automation through distributed applications can increase the efficiency and lower the costs of conducting business between different organizations. For ex-

*Canton is Daml ledger interoperability protocol which abstracts over digital ledgers, allowing to integrate them into a global virtual composable smart contract platform. It demonstrates that a practical system can attain the properties that we deem critical for the overall adoption of smart contract platforms.

ample, electronic price catalogs and order management systems eliminate many labor-intensive manual tasks. Yet, to build an application that spans multiple organizations, rules for exchanging data must be agreed on, correctly implemented, and secured against malicious participants and outsiders. Our thesis is that doing so with existing technology requires too much non-essential technical and organizational complexity, especially when integrating applications. This leads to unnecessary and sometimes prohibitively high costs, as well as brittle systems. Canton aims to significantly lower this complexity, while simultaneously providing scalability and privacy. Developers can then focus on the core business logic and the value-adds instead of the infrastructure and plumbing.

Distributed applications from scratch To build an application that automates a cross-organization workflow, one must devise and correctly implement a set of rules for exchanging data (i.e., a communication protocol). Today's standards help with transporting data (e.g., REST, gRPC) and describing its shape (e.g., XML schemas, session types), as well as handling authentication (e.g., X.509, TLS). But implementing the protocol logic remains a non-trivial task. For example, an estimated 10% of the trade volume in stock markets is subject to manual intervention (reconciliation), due to mismatches and mistakes in interpreting the exchanged data [18]. In response, initiatives such as ISDA's CDM have started specifying the protocol logic alongside the data formats [9]. A *smart contract platform* solves the problem of diverging implementations by providing a shared data encoding and execution logic for the protocol. Of course, this shared logic still has to be correct. In particular, the security of cross-organization distributed applications critically depends on proper authorization. Inadequate authorization features in languages used in existing smart contract platforms resulted in several high-profile incidents, such as the first Parity Wallet attack [15]. Finally, in highly-regulated environments, such as finance, an independently verified log of a participant's protocol interactions is a highly valuable auditing tool. But building such logs is not trivial: a consolidated audit trail for the US stock market has now been in the planning phase since 2012 without an implementation [6].

Integrating applications Business workflows naturally compose into higher-level workflows. Unfortunately, their software implementations usually do not – at least not easily, and not with the desired properties. For example, a travel agency workflow can combine booking a flight with booking a hotel. A desired property of this combination is *atomicity of distributed transactions*: a flight should be booked together with the hotel or not at all. Otherwise, the travel agent must take on the risk of a partially successful booking, increasing the costs for the end customer. But this atomicity is only possible if both the airline and the hotel systems build in specific and compatible support for it. Standards such as X/Open XA [13] exist, but they have to be (correctly!) implemented by all of the involved subsystems, including their off-the-shelf components. This is a significant technical complexity that is not essential to the individual operation,

and thus often never gets implemented [17]. Furthermore, composition often involves delegation of rights. For example, if a travel agency's customer pays using a credit card, (s)he is delegating the control of her/his funds to the agency, by giving away the credit card number. This delegation is intended to be limited: the agent should only get control of the funds needed for the trip purchase. Moreover, there is a link between delegation and atomicity: the funds should be transferred only if the bookings succeed. But today, fund delegation is neither specific nor atomic. The resulting problems again increase costs, from rollbacks to "card not present" fraud that reached US$5.65bn in 2016 [7].

Most smart contract platforms (e.g., [21, 11, 2]) try to address most of the issues raised so far, though their effectiveness in doing so differs as their contract programming languages provide different abstractions. Moreover, the platforms significantly differ in their non-functional properties.

Scalability Platforms relying on proof-of-work blockchains focus on bringing trust in third parties to an absolute minimum. In particular, such a blockchain can yield the digital equivalent of golden nuggets: a bearer token that is neither controlled nor backed by any single real-world party. The platform's contracts can then manipulate the balances in this currency jointly with arbitrary other information. In exchange, the platforms sacrifice scalability: their throughput is typically limited to tens of transactions per second, and the historic data required to use the platform securely often grows unboundedly with time. The throughput can be increased for value transactions by moving them off the blockchain (e.g., Raiden [16] for Ethereum), but this destroys the ability to integrate them with other applications, and introduces problems such as routing and collateralization. But not only blockchains inhibit scaling: most platforms replicate a global shared state at all their participants. This necessarily puts a cap on scaling, as any state change must be processed by all participants.

Privacy A global shared state is also a privacy leak that is unacceptable for use cases such as handling trade secrets, financial data, or healthcare. It also clashes with the data minimization requirements of the European Union's General Data Protection Regulation (GDPR). Pseudonymization helps, but is not a solution [10]. Furthermore, resting the security on a shared global state stored in a blockchain data structure clashes with the GDPR's "right to be forgotten" requirement. Some blockchain-based services have already shut down due to their inability to comply with the GDPR [14]. Advanced cryptography can restore privacy to various degrees, up to full-blown private multi-party computation, but these methods are computationally intensive and present a significant obstacle to scaling. Platforms that are willing to make stronger trust assumptions(e.g., [11, 2]) can limit data visibility without resorting to expensive cryptography. However, limiting visibility can also limit *transparency*: either through error or malice, data might not be distributed to everyone that should see it. This can cause disputes. Finally, the privacy requirements often surpass these platforms' abilities. For example, a stock market trade must happen

atomically, but the buyer's bank should only see the outgoing money transfer, and not what this money bought. All practical platforms promising both such *subtransaction-level privacy* and transparency (e.g., [4]) rely on trusted execution environments, such as SGX [1], for protecting privacy. These protections suffice for some use cases, but recent research [20] makes them inadequate for many others.

Canton Canton handles synchronization, security, and privacy automatically, and lets developers focus on their business logic, written in the programming language Daml [8]. Daml's transaction model makes atomic composition of workflows trivial, and makes authorization and privacy both mandatory, yet simple. The code, written in Daml – though Canton can support any other language with the same transaction model – precisely describes the allowed interactions between the different *parties* of a business workflow. A party can be a legal entity, a physical person, or just one of many accounts for an entity or person. Parties need to make a one-off infrastructure investment to use Canton. Figure 1 illustrates Canton's basic architecture. Parties deploy or connect to one or more Canton *participant* nodes. Each participant node in turn connects to multiple *synchronization domains*, a message sequencing and delivery infrastructure, allowing it to transact with all other parties whose participants connect to some common domain. Once this distributed setup is in place, Canton implements the business logic faithfully, relieving the developer from data transport, atomicity, security, transparency and privacy issues, while providing a verifiable log of all participant actions and allowing for horizontal scaling of independent workflows.

While Canton provides the abstraction of a virtual global ledger, it has no physical global state. A Canton participant only receives, stores, and processes the data it needs to know, where this need is determined at the level of subtransactions. This obeys GDPR's data minimization principles. Canton also provides history pruning and redaction capabilities for its log, allowing a trade-off between the auditability requirements imposed on many financial systems, and the GDPR's right to be forgotten. Canton has no scaling bottlenecks: a participant processes only its own data, and processes workflows synced through different domains in parallel. Also, no expensive cryptography is used on the critical data processing paths. Canton does not aim to eliminate trust in third parties completely, but to keep it small and elective. Third parties run synchronization domains, together with the associated identity management components. Domain operators can see some transaction metadata, but encryption prevents them from accessing the message contents. Domains can also appoint trusted entities that users can selectively involve to help process their workflows even if some of the involved parties are unresponsive. Parties can connect to any domains they trust, as long as the domain operators accept them. That is, domains are permissioned by their operators. However, Canton is permisionless in that anyone can deploy a Canton domain, for any reason. Possible reasons include lowering latency, addressing any throughput bottlenecks due to message

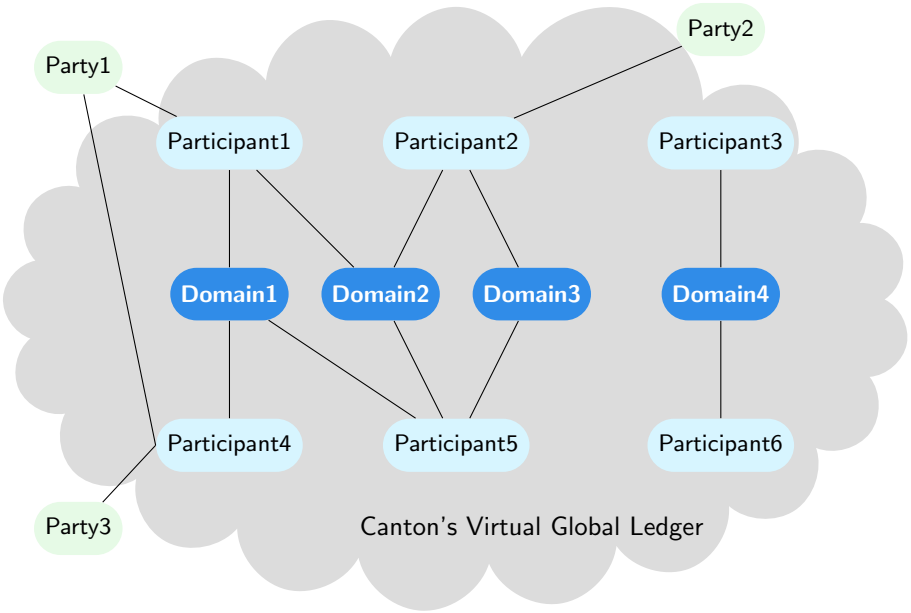


Figure 1: The architecture of Canton. Parties use participant nodes to deploy Canton. Participants connect to other participants through synchronization domains. The infrastructure forms a virtual global ledger where workflows can be composed.

sequencing, or making use of existing PKIs. But the domains are not silos: as long as a common domain for all parties in a transaction exists, parties can still seamlessly and atomically compose workflows running over multiple domains, without any support from the domain operators, avoiding any domain lock-in. Thus, Canton allows truly global workflow composition. We believe that Canton's features, combined together, will enable an unprecedented efficiency in developing distributed applications. In particular, the marginal costs of composing Daml workflows will be a fraction of the integration costs of today's systems.

2 Daml: Authorization, Privacy, Composition

The secret sauce of Canton are Daml [8], Canton's smart contract programming language, and the *hierarchical transactions* that Daml produces. Canton can in general be used with any deterministic language that produces transactions of the same form. However, Daml's features make it particularly suitable for our target setting, where the participants do not have to trust, or even know each other. Daml is a modern functional language featuring a powerful static type system that can rule out many undesired behaviors already at compile

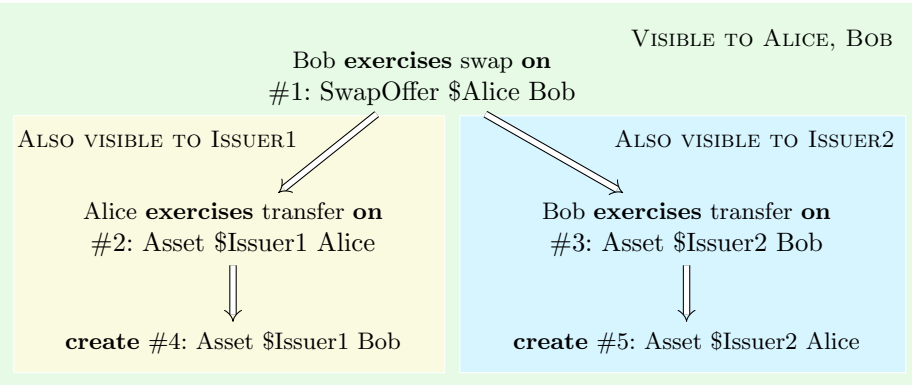


Figure 2: A sample Daml transaction: Bob accepts Alice's offer to swap his asset issued by Issuer2 for Alice's asset issued by Issuer1. Contract identifiers are denoted as #1, #2 etc. Parties have different visibility into the transaction. Outgoing arrows point to the consequences of an action.

time. Such correctness benefits are important for our target setting. Daml has been successfully used to concisely model complex multi-party interactions in multiple domains, including finance [3].

The Daml interpreter deterministically converts a Daml expression into a transaction, and Canton then ensures that the transaction is performed securely and atomically. Daml transactions are hierarchical. A transaction is a list of *actions* on *contracts*. Each action can in turn contain a transaction, a list of *subactions* or *consequences*, giving the transaction its hierarchical structure. Contracts are the data objects of Canton and have unique identifiers. Daml concisely describes all possible actions on a given contract, including their consequences. Figure 2 shows a generic example of a swap transaction. Here, Bob chooses to swap a digital asset that Issuer1 issued to him for a digital asset that Issuer2 issued to Alice. The assets could be anything: digital I-Owe-You tokens, stock certificates, plane tickets and so on. The transaction consists of a single action, which is an *Exercise* action on a *SwapOffer Alice Bob* contract with the identifier 1, performed by the *actor* Bob. The consequences of exercising the swap are two transfer *Exercise* actions on two different *Asset* contracts. Each transfer has a *Create* action as a consequence, creating a new *Asset* contract. An *Exercise consumes* the contract, so that it cannot be exercised again. That is, *double spends* are disallowed. Contracts that have been created, but not yet consumed are called *active contracts*. Active contracts correspond to unspent transaction outputs in Bitcoin.

Daml has a built-in notion of authorization, which is crucial in a distributed setting. Every action has one or more *required authorizers*: parties that have to authorize the action. The authorizers of an *Exercise* are its actors. The authorizers of a *Create* are the *signatories* of the created contract. Signatories are the parties that the Daml programmer annotates as being bound by the

contract's terms, and are marked by the dollar sign in the figure. The terms specify the signatories' real-world obligations. Obligations underlie all cross-entity business workflows in the real world, and in particular form the basis of financial assets. Daml's authorization rules, faithfully implemented by Canton's cryptographic mechanisms, ensure that a party becomes a signatory only voluntarily. Authorization propagates as in the following example. By exercising the swap, Bob authorizes all the direct consequences of the swap, i.e., the transfers. Alice also authorizes them as she is a signatory on the swap offer contract. Thus, both transfers are authorized by their required authorizers. This authorization mechanism yields seamless delegation: Alice and Bob can create a swap offer contract without changing the logic associated with the underlying assets. Notably, this mechanism mimics the offer-acceptance pattern of the legal system: Alice and Bob can become joint signatories on a Daml contract only if the contract creation is a consequence of Bob accepting an offer that Alice signed.

Daml also has a built-in model of privacy. An action on a contract (including its subactions) is visible only to the contract's *stakeholders*. In the example in Figure 2, Bob's exercise of the swap, as well as the *SwapOffer* contract itself are only visible to the stakeholders Alice and Bob. Issuer1 sees only the subtransaction transferring the asset it issued, but nothing about the swap or about Issuer2's asset. Thus, Daml defines its privacy model at the level of subtransactions. Canton faithfully implements both the authorization and the privacy models of Daml.

3 Canton Design Sketch

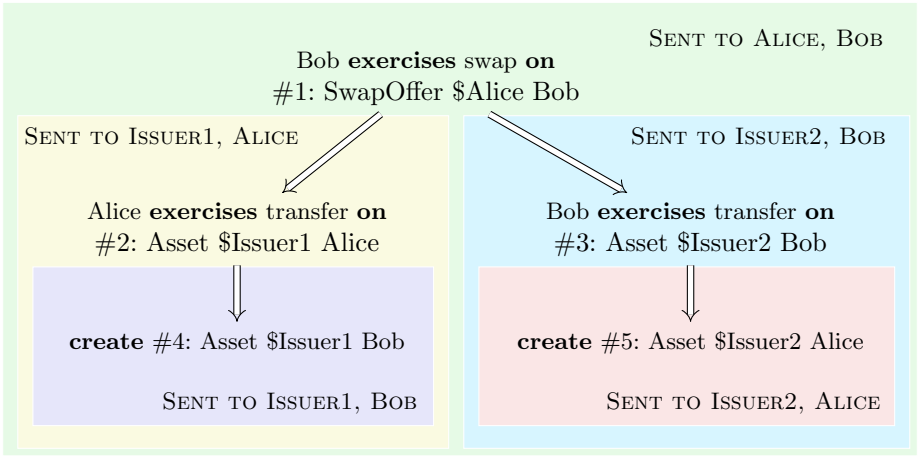
Daml defines an execution model that includes authorization, privacy and prevention of double spends. Canton enforces this model to the letter in a distributed setting. We now explain the main ideas behind Canton.

Distributed consistency At a high level, Canton's primary goal is to provide consistency of shared state across its participants, where the shared state of a set of participants consists of the active contracts on which they are joint stakeholders. The standard approach to distributed consistency is state machine replication [12], where each participant replicates the same global state. It ensures consistency by requiring deterministic transaction processing (i.e., a deterministic state machine) and globally ordered transactions (achieved through a consensus mechanism). Then, each replica applies the transactions locally in the given order, and all replicas reach the same state due to the determinism. This is the conceptual approach taken in Bitcoin and Ethereum. Their proof-of-work consensus mechanisms additionally ensure consistency with almost no trust assumptions, and provide these platforms with trustless bearer tokens. Canton does not aim to create such tokens; instead, it automates the real-world interactions dealing with rights and obligations of real-world parties. Thus, it can take a related, but different technical approach. As with state machine

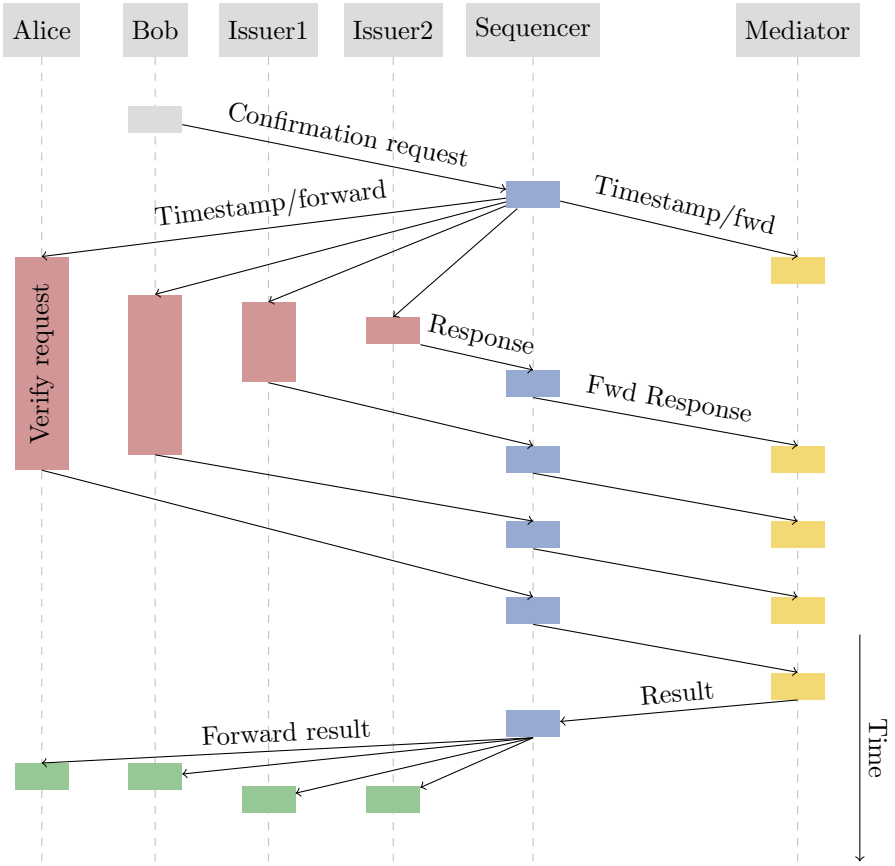
replication, the Canton *messaging service* provides a total order on transaction requests within a domain, and transaction processing is deterministic. However, replicating the entire global state is not acceptable for privacy reasons. Instead, Canton replicates each contract only at its stakeholders. Thus, similar to a sharded database, Canton needs an atomic commit protocol.

Let us consider the swap example again, now shown in Figure 3a. No party involved in the transaction can on their own determine if the transaction is valid. Alice is not a stakeholder on contract #3 and does not know whether it is active. As for Issuer1, the Daml privacy model prohibits it from even seeing which contracts are used by the transaction, other than the assets it issued. Additionally, Canton is built to tolerate malicious participants. Thus, Issuer1 should only accept the transaction if it is sure that Alice authorized the transfer. To solve these issues, Canton processes transactions in two steps. First, the submitter sends a *confirmation request* to every other involved party. The request pertains only to the transaction parts visible to the request's recipient, as shown in Figure 3a. The recipients then check whether the request is valid, and respond with a *confirmation response*. Their checks ensure two things. First, they ensure that the Daml authorization model is respected, and that the correct parties are notified of the transaction, thwarting any malicious behavior by the submitter. Second, they prevent double spends. Double spends are not necessarily a sign of a malicious submitter; they can simply occur under conflicting concurrent workflows. Similar to Calvin [19], the domain's total ordering and Daml's determinism allow everyone to resolve conflicts in the same way, with the difference that Calvin is not built for malicious settings. This is different from a standard two-phase commit, where replicas can always non-deterministically abort a request.

Sample message flow Figure 3b shows a sample Canton message flow. First, the submitter (Bob in the example) splits the transaction up according to the Daml privacy model. Next, Bob sends the chopped-up transaction as a batch of messages to the domain's *sequencer*. This batch is the confirmation request. Crucially, the messages are encrypted with the recipients' keys, so the sequencer does *not* learn the message content. It simply orders, timestamps and distributes the individual messages from the batch. The timestamps grow monotonically, yielding a total order, as all messages on a domain go through the sequencer. The sequencer is trusted to deliver all messages according to this order (this trust could be distributed through Byzantine fault tolerant mechanisms if necessary). Then, the recipients verify their parts of the confirmation request. They attempt to *lock* each contract consumed by their part of the transaction. If the verification fails, because the submitter is malicious, or because the lock is already taken, the verifier sends a negative response. Otherwise, it sends a positive one. All responses are signed and sent to the *mediator*, an entity run by the synchronization domain. Canton uses cryptography to ensure that the responses can be tied into a coherent whole and that malicious participants cannot violate the system's integrity. The mediator aggregates the responses



(a) A Canton transaction.



(b) Canton message flow.

Figure 3: Canton's transactions and message flow.

into a final result, signs it, and sends it to the transaction's participants. The mediator serves two purposes. First, it protects the participant's privacy. For example, Issuer1 does not learn that Issuer2 is also involved in the transaction. Second, while privacy could also be preserved by pseudonymizing the parties involved, the mediator lowers the total number of messages that must be exchanged. After receiving the mediator's result, the participants release the locks and, if the result is positive, apply the transaction. That is, they consume the exercised contracts and create new ones as specified.

Aborts and liveness The above scheme has the problem that a transaction can get stuck if one of the participants does not respond, because of failures or maliciousness (mediators are assumed to be highly available). The contracts that the transaction consumes would then remain locked until a response is received. We resolve the problem by making the locks abortable, but only after a specified lease time, known domain-wide. All time-related reasoning is based on the sequencer timestamps: any transaction participant may send an abort after observing a sequencer timestamp exceeding the lease time, making the contract available for other transactions.

The unresponsive participant can still violate *liveness* for a transaction: it can prevent even conflict-free transactions from ever getting accepted. This violation is sometimes acceptable. For example, if a transaction creates a digital airplane ticket, accepting the transaction would be of dubious value if the airline's systems stay unresponsive. For cases where liveness is nevertheless paramount, a Canton domain can use *attestators*, entities declared as trusted for confirmations. The transaction's parties can choose to disclose sufficient history to the attestator to convince it that the contracts involving unresponsive parties are still active. The history consists of two parts. The first part are inclusion proofs, which prove that the contract was recently active. This proof relies on signed fingerprints of shared state snapshots, which are regularly exchanged between all pairs of parties that share contracts. The other part are exclusion proofs, showing that the contracts have not been consumed since the last snapshot. As an alternative to attestators, the parties can choose to involve *VIP participants* as stakeholders in some of their contracts. Like attestators, these participants are trusted for confirmations. However, like all stakeholders, and unlike attestators, they see all actions involving the contracts in question.

Auditability and GDPR Daml contracts can have real-world agreements associated with them. For any contract created or consumed by a transaction, Canton provides cryptographic evidence of how this happened. Since the Daml authorization model follows the principles of contract law, the *non-repudiation* property of this evidence can be useful in a real-world dispute. Furthermore, the sequencer signs the stream of messages that it sends to any of the participants, which serves as a verifiable log. As the participant's prescribed behavior is deterministic, the actual behavior can later be audited both for correctness of operation and the content of the exchanged data. This makes Canton suitable

for regulated environments, such as financial markets. However, as Canton's commit protocol does not use the log in any way, the history can be pruned or moved to cold storage at any time.

As Figure 3a shows, Canton strongly aligns with the data minimization requirements of GDPR. As data is processed only where needed, it also aligns with the requirements on locality of data processing. However, the second main principle of the GDPR, the right to forget, clashes with the non-repudiation and auditability requirements. To resolve this tension, Canton allows the participants to replace a log entry, or a piece of evidence, with a new one. The replacement must be signed by all the other participants involved in the old entry. The replacement refers to the old data only by a hash. In Canton, all hashes are salted. Once the old data entries are deleted, so are their salts, and the hashes become unlinkable to the original data. In addition to contracts and logs, personal information can also appear in the data used by the identity management component of Canton. However, Canton takes care to separate such information into *identity metadata*, which can always be dropped without affecting auditability.

Multiple domains and global composability A single Canton synchronization domain allows atomic composition of arbitrary workflows. However, having multiple domains can be beneficial for a multitude of reasons. For example, a single global domain would impose a high communication latency for participants located far from the sequencer. Multiple domains can also help to increase throughput: requests from different domains can be processed completely in parallel. There might also be operational concerns. For example, for critical workflows, a new domain with restricted access can be spawned, to protect from denial of service and similar attacks. Additionally, we saw that a domain can specify entities that are trusted for confirmations, but any set of such entities is unlikely to be universally trusted by everyone. Finally, the domain operator might charge for its services, and a single global domain would then lock everyone into using it.

Having multiple domains, however, opens up a new challenge, of how to compose workflows across domains. In Daml terms, composing workflows specifically means that contracts created on different domains can be used within a single transaction. Without such an ability, the synchronization domains would not truly solve the composability problem: they would just create bigger silos. The hard part is guaranteeing atomicity of such transactions, while maintaining resilience to unresponsive participants. Since different domains have no common notion of ordering on its messages, reconciling atomicity with resilience can become impossible. Thus, Canton allows cross-domain transactions whenever there exists a single domain that all participants in a transaction are connected to. Furthermore, Canton allows transfers of contracts between domains. A contract transfer simply marks a different domain as the new authority for ordering actions on the given contract.

	Apps on DBs	Ethereum	Fabric	Corda	Daml on one DB	Canton
Distributed transactions	−−	++	++	++	++	++
Robust authorization	−−	−	−	o	++	++
Transaction privacy	++	−−	−	o	+	++
Subtransaction privacy + transparency	++	−−	−−	−−	+	++
GDPR compliance	++	−−	−−	o	+	++
Horizontal scalability	++	−−	++	++	++	++
Global composability	−−	++	−−	+	−−	++

Table 1: Comparison of Canton with other smart contract platforms and traditional tech. Scale, from worse to better: −− − o + ++

4 Comparison to Other Platforms

We conclude by comparing Canton with several other prominent smart contract platforms. We also compare it with a traditional setup of organizations implementing applications which run on isolated databases, and a solution where Daml defines the business logic but is executed on top of a single, centralized database instance, operated by a trusted third party. The comparison is from the perspective of implementing a cross-organization distributed application. Table 1 shows an overview of how the platforms fare with respect to the properties discussed in this whitepaper.

Distributed transactions and robust authorization All smart contract platforms support distributed transactions out of the box, as opposed to a traditional setup of isolated databases and applications sitting on top of them. However, we see significant differences when it comes to handling authorization. In Ethereum and Hyperledger Fabric [2], a contract's programmer has access to the submitter's identity, but must manually add authorization checks where needed. This is a fragile mechanism, as witnessed by the authorization flaw in the Parity Wallet [15], a contract written by the core developers of Ethereum. In Corda, authorization is much more fine-grained. Each action (command, in Corda parlance) within a transaction has an associated set of parties that have signed, and thus authorized this action. The contract writer can then check that these authorizations suffice for the task at hand. However, delegation is hard in Corda: developers must manually decide which commands they choose to sign. That is, instead of simply signing the *SwapOffer* contract like in the Daml example shown earlier, each party must sign the appropriate transfer.

(Sub)transaction privacy A traditional setup provides both transaction and subtransaction privacy since all connections are bilateral. Ethereum is the complete opposite: it uses a global shared state visible to everyone. Thus, all transaction details are public. Fabric's default privacy model is similar to

running many instances of Ethereum in parallel. Each such instance is called a channel, with all channel data visible to all channel members. Channels are silos, in that there are no atomic transactions across them. Thus, users must make a choice between atomicity and privacy. Alternatively, users can store only data fingerprints on the ledger, using Fabric's so-called private data collections. However, both transparency and the ability to atomically compose workflows across such collections are then lost. Corda, in contrast, limits distribution of a new transaction to the parties involved in it, and to special *notary* nodes, which are part of the infrastructure. However, there is no subtransaction-level privacy. That is, in the swap example, the issuers would learn the full details of the swap. Furthermore, for every contract used in the transaction, the entire chain of transactions leading up to this contract is shown to everyone involved in the transaction. That is, the Corda model would reveal to Bob every transaction leading up to contract #2 in Figure 2. Corda proposes to protect privacy using Intel's trusted hardware execution environment, SGX [1]. But recent research results [20] would enable a malicious Bob to easily recover the transaction history without leaving a trace, even when SGX is used. The reliance on SGX can thus be problematic for many use cases, also in other platforms [5, 4]. Daml on SQL, where the parties use Daml to talk to a single SQL database that stores all data, would provide subtransaction-level privacy for the parties. However, the database operator(s) would have access to all data. In Canton, the infrastructure nodes only see encrypted data, thus learning nothing about it except for the size.

GDPR compliance For GDPR compliance, applications must be designed to minimize data collection regardless of the underlying technology. But even when the application is compliant, sharing the data with the entire world (as in Ethereum), or indiscriminately with all channel members (as in Fabric) violates the compliance. Furthermore, the underlying technology makes a large difference in implementing the right to forget, as this requires amending history. While this is straightforward with a traditional setup, it is impossible in systems with an immutable history, such as the one provided by the hash chains of Ethereum and Fabric. Corda largely complies with data minimization requirements and does not rely on hash chains, but it still verifies the entire history of every contract in all transactions. Thus, implementing the right to forget compliance would be difficult, as contracts would have to be either returned to the issuer and reissued, or frozen until their entire history is purged of the personal data.

Scaling and composability Horizontal scaling (for unrelated workflows) requires some form of partitioning. All of the considered technologies enable partitioning, apart from Ethereum, which uses a global state. However, the situation looks very different when we consider global composability. Globally composing applications running on databases boils down to distributed transactions and authorization. Fabric provides no atomic composition across different

channels. Similarly, running Daml on SQL also provides no way to compose two such installations on two databases. In Ethereum, global composability is trivial, since the state is global. However, composability is lost if multiple instances of Ethereum are deployed, such as in Enterprise Ethereum scenarios. Corda's transfer protocols can be used for atomic composition, by first transferring all contracts from their original notaries to a single destination notary. Canton also supports transfers, and it supports direct atomic composition, as long as a joint domain exists.

References

[1] Ittai Anati et al. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. ACM New York, NY, USA, 2013.

[2] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 30.

[3] Digital Asset. *Daml examples*. GitHub. URL: `https://github.com/digital-asset`.

[4] Mic Bowman et al. "Private Data Objects: an Overview". In: *arXiv:1807.05686 [cs]* (July 16, 2018). arXiv: `1807.05686`. URL: `http://arxiv.org/abs/1807.05686`.

[5] Marcus Brandenburger et al. "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric". In: *arXiv:1805.08541 [cs]* (May 22, 2018). arXiv: `1805.08541`. URL: `http://arxiv.org/abs/1805.08541`.

[6] *CAT NMS Plan The Consolidated Audit Trail*. URL: `https://www.catnmsplan.com/home/about-cat/cat-nms-plan/index.html`.

[7] HSN Consultants. *Nilson Report*. Oct. 2016. URL: `https://nilsonreport.com/upload/content_promo/The_Nilson_Report_10-17-2016.pdf`.

[8] *Daml - The Enterprise Smart Contract Programming Language*. URL: `https://daml.com/`.

[9] *Digital Asset and ISDA Introduce Tool to Help Drive Adoption of ISDA CDM*. International Swaps and Derivatives Association. Apr. 10, 2019. URL: `https://www.isda.org/2019/04/09/digital-asset-and-isda-introduce-tool-to-help-drive-adoption-of-isda-cdm/`.

[10] Steven Goldfeder et al. "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies". In: Privacy Enhancing Technologies. 2018, p. 21.

[11] Mike Hearn. *Corda: A distributed ledger*.

[12] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[13] X/Open Company Ltd. *X/Open CAE Specification: Distributed Transaction Processing: the XA Specification*. X/Open Company Limited, 1991.

[14] *PICOPS to be discontinued on May 24th, 2018*. Blockchain Infrastructure for the Decentralised Web. May 18, 2018. URL: https://www.parity.io/picops-discontinued-may-24th-2018/.

[15] Santiago Palladino. *The Parity Wallet Hack Explained*. Zeppelin Blog. July 19, 2017. URL: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

[16] *Raiden Network*. URL: https://raiden.network/.

[17] Prabhu Ram, Lyman Do, and Pamela Drew. "Distributed Transactions in Practice". In: *SIGMOD Record* (1999), p. 7.

[18] Goldman Sachs. *Profiles in Innovation Blockchain*. 2016. URL: https://github.com/bellaj/Blockchain/blob/master/Goldman-Sachs-report-Blockchain-Putting-Theory-into-Practice.pdf.

[19] Alexander Thomson et al. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 1–12.

[20] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 991–1008.

[21] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.