APPENDIX

A. HYPERLEDGER CONCEPT MAPPING

Hyperledger Fabric is a popular opensource consortium blockchain implementation promoted by IBM. In Fabric, every blockchain node is called a peer. A peer joins different *channels* with permissions, where each channel keeps its own sub-ledger. Moreover, each *channel* manages two data stores. One is a ledger that stores all the transaction logs. This store is imutable and non-repudiable. Every transaction in the ledger is signed by both the submitter and the endorser peers. The other is a state database, the default implementation of which is CouchDB, a NoSQL JSON document store. The state database persists the current state of data objects. A transaction, such as a payment, generates a record in the ledger and affects the data objects, such as the account balance, in the state DB. A state database can be recovered by playing back transactions in the ledger store from the genesis block. In Fabric, a transaction submission is divided into two phases. In the first phase, an application sends a transaction proposal to one or more peers, as endorsers, for evalutation. This step will not impact the actual data store of the peers. Instead, it returns the evaluation result in a ReadWriteSet that contains all data objects that are read from and would be written to the state database during the transaction. Only when all the endorsers agree with the same ReadWriteSet can the client enter the second phase to send the transaction to the ordering service for eventual commit. The ordering service broadcasts the transaction with a deterministic order to all peers in the channel. These peers update both the raw ledger and the state database according to the ReadWriteSet associated with the transaction. Remarkably, for concurrent transactions, the endorsement phase is executed concurrently while the commit phase is strictly executed sequentially according to the order given by the ordering service. Moreover, for the execution of a transaction, there is a time window between the two phases. If any data objects in the ReadWriteSet are modified by some other transactions during this time window (*i.e.*, the data is dirty), the commit phase fails and the transaction is discarded even though it was endorsed.

Fabric comes with a smart contract development framework named Composer. It builds a natural mapping from Fabric to a collaborative network. In Composer, all elements of a collaborative network are modelled as either participants, assets, or transactions. A participant in Composer can be either a core participant, a sub-participant or a satellite participant. Data classes map to asset definitions. The smart contract terms are implementated using the JavaScript language.

B. DISTRIBUTED TRANSACTIONS

X/Open XA [1] uses a lock coordinator strategy. It has become a de facto standard in behavior of transaction model components for many commercial databases and applications. In an XA implementation, the transaction manager commits the distributed branches of a global transaction by using a two-phase commit (2PC) protocol. In phase one, the transaction manager directs each resource manager to prepare to commit, which is to verify and guarantee it can commit its respective branch of the global transaction. If a resource manager cannot commit its branch, the transaction manager rolls back the entire transaction in phase two. In phase two, the transaction manager either directs each resource manager to commit its branch or, if a resource manager reported it was unable to commit in phase one, the global transaction is rolled back.

Deterministic transactions [2] are based on a deterministic database. The basic idea is to divide the causes of transaction failure into *non-deterministic events* (such as node failures) and *deterministic events* (such as transaction logic that forces an abort or ReadWriteSet inconsistency in our case). All *deterministic events* should be ascertained before the commit. All *non-deterministic events* can be fixed via retrying. Therefore, as long as the commit request is sent, the transaction will succeed in the end and thus no rollback is needed. In Fabric, all logic errors in smart contracts are returned during the endorsement phase. In the commit phase, the ordering service is the place where the actual commit command is broadcast to peers. It can see the complete transaction history and the corresponding ReadWriteSets of the subledgers it manages. Before the broadcast of a committed transaction, it can verify that all the endorsers provided the same ReadWriteSet and, more importantly, accurately predict if the data objects in the current ReadSet are made dirty due to historical transactions. If no *deterministic event* is detected, it can safely broadcast the commit message and assume the commit success. This approach is more efficient since there is no lock during the commit phase. However, it is not always true that all *deterministic events* could be figured out before the commit phase. For example, in Fabric different subledgers can choose to have different ordering services which do not communicate with each other. If a distributed transaction writes data objects to two subledgers that use different ordering services, neither ordering service can ensure if there is *deterministic event*, such as dirty ReadSet, for the other subledger which lead to a failure.

C. FABRIC XA

As discussed in Appendix A, there is a 2PC-like process during the transaction submission of Hyperledger Fabric. However, this process is designed for reaching consensus among counterparties. No guarantees are given to ensure consistency when a global transaction is split into multiple branch transactions of different subledgers. We extend Hyperledger Fabric to support XA.

Figure 1 shows the message flow during a sample distributed transaction in Fabric. In order to achieve a distributed transaction, all branch transactions' locks on respective subledgers must be held for the full duration of transaction committing. As long as one or more subledger reports an error, all other subledgers will "roll back" changes. In a fully untrusted environment, additional actions such as zero knowledge proof [3] can be taken during the business consensus phase to balance privacy and equity.

Algorithm 1 shows the process of 2PC for Fabric. Compared with a DBMS, Fabric provides no pre-commit, rollback capability, as it is immutable. All pre-commit/ commit/ rollback operations are considered as regular Fabric transactions, which involve endorse phase and (Fabric) commit phase. In the pre-commit phase (line 1-15), a global transaction t is split to branch transactions $\{t'\}$. Each branch transaction is sent to the corresponding subledger. Instead of making actual changes to data objects, the ReadWriteSet is directly committed as data to store. At the same time,

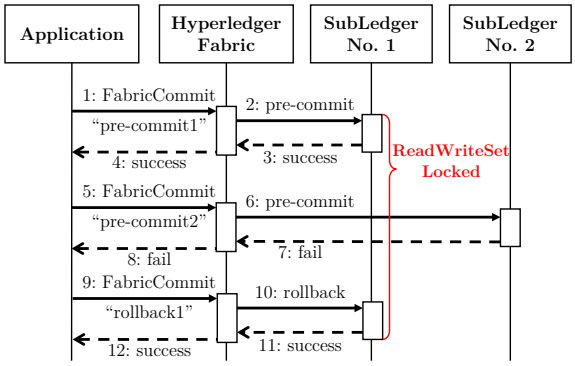

Figure 1: A rollback instance for Fabric XA

all data objects specified in the ReadWriteSet are locked. Any other transactions, except the corresponding 2PC commit/rollback transaction that read or write to the locked data objects are discarded during the lock period. Any pre-commit failure will result in rollback for the pre-committed subledgers. Then in the (2PC) commit phase (line 16-19), data objects are unlocked and changes are committed to the corresponding subledgers.

Algorithm 1: Two-Phase Commit for Fabric

```
1  Pre-commit Phase:
2  Endorse transaction t, and get the global ReadWriteSet
     RW_t;
3  Split RW_t to {RW_t'} according to subledgers;
4  for each RW_t' do
5      Construct a branch transaction t' where RW_t' is
         considered as a data object to be committed;
6      Endorse t';
7      if endorse failed then
8          abort globally;
9  for t' ∈ {t'} do
10     Commit t' to corresponding subledger and lock data
         objects in write set W_t';
11     if commit failed then
12         for each committed branch tx t'' do
13             Construct a rollback transaction r'' that
                 unlocks all locked data objects;
14             Endorse and commit r'';
15         abort globally;
16 Commit Phase:
17 for t' ∈ {t'} do
18     unlock the data objects;
19     commit according to the ReadWriteSet;
```

D. PROOF OF THEOREM 1

Proof. We prove a GEP problem NP hard by reducing the Minimum Vertex Cover (MVC) problem to a GEP instance. An instance of MVC is a vertex-weighted graph $G =< V, E, W >$ the objective of which is to find a vertex cover O that minimizes $\sum_{v_i \in O} w_i$. We construct a data class set C. Each data class corresponds to a vertex $v_i \in V$ with a weight value of $w_i \in W$. For each $e =< v_i, v_j >\in E$, construct a transaction t with $f_t = 1$ and $W_t = \{c_i, c_j\}$. Add a new participant p_{t,c_i} to c_i's read group R_{c_i} and p_{t,c_j} to R_{c_j}. In this way, c_i and c_j has different read groups. Next, for each $c_i \in C$, we set $N_{sq} = \frac{w_i}{|R_{c_i}|} \times \Pi_{c_j \in C} |R_{c_j}|$ and $cost_{enc}(c_i, t) = N_{sq}$ for any tx t involving c_i.

At last, we set $\alpha_O = 0$, and $\alpha_l = 1 + \sum_{c_i \in C} w_i \times \Pi_{c_j \in C} |R_{c_j}|$, otherwise. The ledger partitioning overhead becomes

$$cost_{dt}(t) = \begin{cases} 0 & |f_{RG}(W_t - O)| \leq 1 \\ \alpha_l \times |f_{RG}(W_t - O)| & otherwise \end{cases}$$

, where O is the encrypted subledger. Under this setting, $\mathcal{C}_D$ has to be either 0 or greater than α_l. However, if $\mathcal{C}_D > \alpha_l$, we can always find a solution $O = M$ whose overall cost is $\alpha_l - 1$ which is smaller than $\mathcal{C}_D$. Therefore, an optimal solution O has to make $\mathcal{C}_D$ to 0. The objective function of GEP becomes $\mathcal{C}_E = \sum_{t \in T} \sum_{c \in RW_t \cap O} cost_{enc}(c, t) \times f_t$. Solving the constructed GEP problem results in a data class set O that minimizes

$$\sum_{c \in O} \sum_{t \in T \wedge c_i \in RW_t} cost_{enc}(c_i, t) = \sum_{c_i \in O} w_i \times \Pi_{c_j \in C} |R_{c_j}|$$

, which minimizes

$$\sum_{c_i \in O} w_i.$$

Moreover, since the read groups of all the data classes are different, $\mathcal{C}_D = 0$ only if $\forall t \in T$, $|f_{RG}(W_t - O)| \leq 1$. In the other words, O is a minimal cover for $\{W_t\}$ and corresponding edge set E. □

E. CASE STUDY: SUPPLY CHAIN FINANCE

A Chinese provincial supply chain management monopoly planned to establish a supply chain finance league consisting of a controlled number of financing companies and hundreds of suppliers and manufacturers. Our proof-of-concept system with this company built an end-to-end blockchain supply chain network for financing based on Hyperledger Fabric. In this scenario, a manufacturer signs a trade contract with a supplier. The two parties complete the trading process and during the payment a financing company is involved to provide financing service to the manufacturer. The trading details should be kept within the three participants plus the platform operator acting as notary. The whole process involves 530 blockchain queries and 235 blockchain transactions triggered by the three parties. The initial version of the system did not adhere to the *confidentiality* contraint of the DDAC problem. An unauthorized core participant could read all trading data in the network directly from the peer's backend state DB. We applied the proposed DDAC framework to the system. The performance test was conducted on the same environment as described in the experiment. We run the queries and transactions for the end-to-end scenario sequentially without any artificial delays. The initial version without DDAC control took 1475 seconds in average to complete the performance test. When the DDAC framework was applied, it took 2253 seconds in total. 34.5% performance overhead was observed, which is acceptable in practice.

6. REFERENCES

[1] Distributed Transaction Processing: The XA Specification. `http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf`, 1991.

[2] A. G. Thomson. *Deterministic Transaction Execution in Distributed Database Systems*. PhD thesis, New Haven, CT, USA, 2013. AAI3578463.

[3] H. Wu and F. Wang. A survey of noninteractive zero knowledge proof system and its applications. 2014:560484, 05 2014.