Endorsement in Hyperledger Fabric via service discovery

Y. Manevich
A. Barger
Y. Tock

Hyperledger Fabric (HLF) is a modular and extensible permissioned blockchain platform. The platform's design exhibits principles required by enterprise-grade business applications, such as supply chains, financial transactions, asset management, etc. For that end, HLF introduces several innovations, two of which are smart contracts in general-purpose languages (chaincode in HLF), and flexible endorsement policies, which govern whether a transaction is considered valid. Typical blockchain applications comprise two tiers: The "platform" tier defines the data schema and embedding of business rules by means of chaincode and endorsement policies; the "client-side" tier uses the HLF software development kit (SDK) to implement client application logic. The client side should be aware of the deployment address of chaincode and endorsement policies within the platform. In past releases, this was statically configured into the client side. As of HLF v1.2, a new feature called service discovery*, presented in this paper, provides APIs that allow dynamic discovery of the configuration required for the client SDK to interact with the platform. This enables the client to rapidly adapt to changes in the platform, thus improving the reliability of the application layer and making the HLF platform more consumable.*

1 Introduction

Blockchain technology is gaining a lot of traction, becoming one of the most appealing and intriguing areas of interest for both research communities and industrial parties. The popularity of blockchain technologies stems from its huge potential of developing a wide range of distributed applications, allowing safe collaboration between mutually distrusting parties, without the use of a central trusted authority.

Blockchain could be viewed as an append-only immutable data structure—a distributed ledger that maintains transaction records between distrusting parties. The transactions are usually grouped into blocks. Then, every party involved in the blockchain network takes part in a consensus protocol to validate transactions and agree on an order between blocks, consequently building a hash chain over these blocks. This process forms a ledger of ordered transactions and is crucial for consistency and integrity. Each party is responsible, maintaining its own copy of the distributed ledger, not assuming trust on anyone else. Therefore, blockchain protocols are related to Byzantine fault-tolerant consensus.

Much of the increasing enthusiasm around Bitcoin [1] is attributed to blockchain as a promising technology to run trusted exchanges in the digital world. Bitcoin is operated in public, where anyone can join or leave the blockchain network, and no one is required to specify the real identity. Such blockchain systems are known as public or permissionless blockchains. Public blockchains inherently involve the notion of a native cryptocurrency and are mostly based on the proof-of-work consensus protocol to compensate for the lack of identity in the open-group model. The proof-of-work consensus protocol has several salient disadvantages, which are as follows:

1) a huge computational cost that manifests in prohibitive power consumption;
2) probabilistic nature of transaction confirmation, leading to large confirmation latency;
3) low transaction throughput.

These factors make public blockchains unsuitable for enterprise-grade application. Therefore, growing interest from industry triggered the development of new blockchain platforms designed for permissioned settings, where the blockchain protocol runs among a set of known,

Digital Object Identifier: 10.1147/JRD.2019.2900647

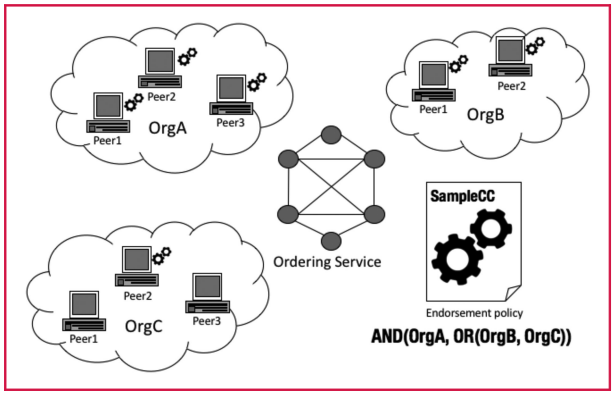

Figure 1

High-level structure of Hyperledger Fabric blockchain network. Presented is a deployment that includes three organizations **OrgA**, **OrgB,** and **OrgC**, each including three, two, and three peers, respectively. The chaincode **SampleCC** is deployed on some of the peers, and its associated endorsement policy requires the signatures of at least one peer from **OrgA**, and at least one peer form either **OrgB** or **OrgC**. The ordering service is responsible for the total order of transactions.

authenticated participants. This is a natural evolution to address requirements posed by business applications running blockchain among a set of identifiable participants that do not fully trust each other.

It is possible to embed business rules into a Turing-complete programmable transaction logic, to be executed by blockchain in the form of a *smart contract*, as introduced by Ethereum [2]. The Bitcoin script was a predecessor of this concept allowing the transfer of native crypto-coins (bitcoins) from one owner to another. A smart contract provides an abstraction that resembles the functionality of a trusted distributed application, leveraging underlying blockchain facilities to gain security and consistency guaranties. Many permissioned blockchains use a replicated state machine [3] paradigm: They order the transactions and then execute them on all peers. This is known as the *order-execute* architecture that leads to intolerance to nondeterministic smart contracts and to sequential execution of transactions that severely limits performance [4].

Hyperledger Fabric [4] (HLF) is an open-source project, released by the Linux Foundation. It introduces a new architecture (see **Figure 1**) for enterprise-grade permissioned blockchain platforms following the novel paradigm of *execute-order-validate* for distributed execution of smart contracts (*chaincode* in HLF). In contrast to the *order-execute* paradigm, in HLF, transactions are first tentatively executed by a *subset* of peers (endorsed). Transactions (with tentative results) are then grouped into blocks and *ordered*, and finally a *validation* phase makes sure that transactions were properly endorsed and are not in conflict with other transactions. Validated transactions are then committed to the blockchain state. This architecture allows multiple transactions to be executed in parallel by disjoint subsets of peers, increasing throughput, and tolerates nondeterministic chaincode. Invalid transactions are dropped in the validation phase. The *endorsement policy* is the set of rules that determines which subset of peers should execute a transaction and what constitutes a valid execution. In a sense, HLF benefits from the combination of two well-known approaches for replication, passive and active [5, 6].

Blockchain applications are typically composed of two tiers. The first—called the "platform tier"—focuses on modeling the data schema and embedding of business rules into the blockchain by means of chaincode and endorsement policies. The second—called the "client tier"—uses the software development kit (SDK) provided by HLF to implement client-side application logic.

For its proper operation, the client needs to know the identifier and deployment address (i.e., peers) of the chaincode it intends to invoke. It also needs to select a proper subset of those peers in order to fulfill the endorsement policy coupled with said chaincode. The challenge is that all of these entities are dynamic: Peers may be added, removed, or simply crash; endorsement policies may be updated; and chaincode may be upgraded.

In past releases of HLF (before v1.2), the chaincode identifier and location, as well as endorsement policies, were statically configured into the HLF client. That is, the client was statically configured with the addresses of the peers that need to execute and endorse a transaction proposal for a particular chaincode. This limited the reliability and availability of the client in the event of changes in the platform. Moreover, client configuration was complicated and technical, which made the platform more difficult to use.

In this paper, we describe the design and implementation of the *service discovery* component, introduced in HLF v1.2, which addresses the challenges posed by the dynamic nature of the platform. Service discovery provides APIs that allow the client application to dynamically discover the configuration details of the endorsement policies and chaincode it needs to use. It, therefore, alleviates the client application developer from the burden of painstakingly reconfiguring the client every time these change, thus increasing the availability and resiliency of the client-side applications. Service discovery leverages the membership propagation capability of the HLF replication layer [7] to gather and disseminate the necessary information needed to implement these APIs.

The remainder of this paper is organized as follows: Section 2 provides some background by briefly describing the internal structure of HLF. Section 3 describes the HLF gossip layer, whereas Section 4 outlines the endorsement

policies. Next, Section 5 presents the design and implementation of the new *service discovery* component, and finally, Section 6 concludes this paper.

2 Background

Prior to HLF, all blockchain platforms, permissioned or permissionless, followed the order-execute pattern. That is, network participants use a consensus protocol to order transactions, and only once the order is decided, all transactions are executed sequentially, thus essentially implementing active state machine replication [3]. The order-execute approach poses a set of limitations. The fact that transactions have to be executed sequentially effectively leads to throughput degradation, becoming a bottleneck. In addition, an important issue to consider is the possible nondeterministic outcome of transactions. The active state machine replication technique implies that transaction execution results have to be deterministic in order to prevent state "forks." Most of the current blockchain platforms implement domain-specific language to overcome the problem of nondeterminism [8].

HLF provides a modular architecture and introduces a novel *execute-order-validate* approach to address the limitations of the *order-execute* approach mentioned before. A distributed application in HLF is basically composed of the following two main parts (see Figure 1).

1) *Chaincode*: It is business logic implemented in a general-purpose programming language (Java, Go, and JavaScript) and invoked during the execution phase. The chaincode is a synonym for the well-known concept of smart contracts and is a core element of HLF, which is executed in a distributed fashion.
2) *Endorsement policies*: They are rules that specify what is the correct set of peers responsible for the execution and approval of a given chaincode invocation. Such peers, called endorsing peers, govern the validity of the chaincode execution results by providing a signature over those results. The endorsement policies are defined with logical expressions such as: **AND(OrgA, or(OrgB, OrgC))**.

2.1 Node types

The HLF blockchain network is formed by nodes that could be classified into the following three categories based on their roles.

1) *Clients*: Clients are network nodes running the application code, which coordinates transaction execution. Client application code typically uses the HLF SDK in order to communicate with the platform.
2) *Peers*: Peers are platform nodes that maintain a record of transactions using an append-only ledger and are responsible for the execution of the chaincode and its life-cycle. These nodes also maintain a "state" in the form of a versioned key-value store. In order to allow load balancing, not all peers are responsible for execution of the chaincode, but only a subset of peers called *endorsing peers*.
3) *Ordering nodes*: Ordering nodes are platform nodes that form a cluster that exposes an abstraction of atomic broadcast in order to establish total order between all transactions. Ordering nodes are completely oblivious to the application state and do not take any part in transaction validation or execution.

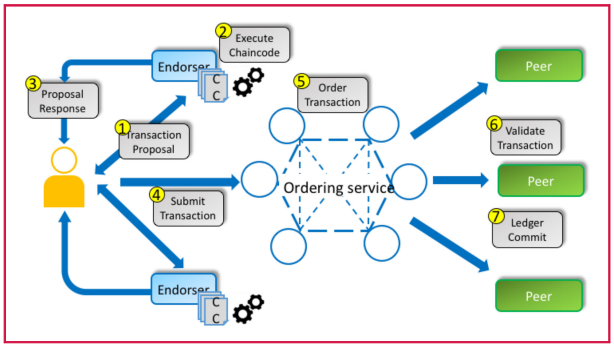

Figure 2

Hyperledger Fabric—high-level transaction flow. Client (yellow actor) proposes a transaction to the endorsing peers (blue) and collects transaction responses. Client then submits a transaction to the ordering service, which orders incoming transactions and cuts them into blocks. Peers (green) pull blocks from the ordering service, validate the transactions, append them to the ledger, and apply valid transactions to the state.

In order to provide finer grained privacy and confidentiality, HLF introduces the concept of *channels*, a high-level abstraction that basically represents an isolated blockchain network. Each channel can contain different or even disjoint sets of peers, thus allowing to segregate application state achieving greater privacy by partitioning data across different nodes.

2.2 Transaction execution flow

The following summarizes the execution flow of a transaction submitted by a client into HLF (see **Figure 2**).

1) The client uses an SDK to form a *transaction proposal*, which includes the channel name, the chaincode name to invoke, and the input parameters to the chaincode that is about to be executed. Next, the client sends the transaction proposal to all endorsing peers to satisfy the endorsement policy of the given chaincode.
2) Endorsing peers simulate the transaction based on the parameters received from the client. The endorsing peers invoke the chaincode, record state updates, and produce output in the form of a

versioned read–write set. The state does not change at this stage. Next, each endorsing peer signs the read–write set and returns the result back to the client.

3) The client collects responses from all endorsing peers and validates that results are consistent, i.e., enough endorsing peers have signed the same payload.
4) Then, client concatenates all the signatures of the endorsing peers along with the read–write sets, creating a transaction that is submitted to the ordering service.
5) The ordering service collects all incoming transactions, orders them to impose total order of transactions within a channel context, and periodically cuts blocks that include all those transactions ordered.
6) For each organization, a single peer pulls new blocks from the ordering service and disseminates them by using scalable middleware for ledger replication, whose implementation is based on an epidemic diffusion-based protocol—gossip [7]. Upon receiving a new block, each peer iterates over the transactions in it and validates the following: first, the endorsement policy, i.e., whether the set of endorsing peers' signatures satisfies the endorsement policy correlated to the chaincode; and, second, performs multiversion concurrency control checks against the state.
7) Once the transaction validation is finished, the peer appends the block to the ledger and updates its state based on valid transactions. After the block is committed, the peer emits events to notify clients connected to it.

3 Gossip layer

One of the immediate benefits of the HLF architecture is the ability to independently scale each of the *execute-order-validate* phases. However, the fifth step of the transaction execution flow (see Section 2.2)—block dissemination to peers—poses additional challenges. Most consensus algorithms (both BFT and CFT) are very sensitive to the available bandwidth. Therefore, the ability to scale the ordering service is limited by the network capacity of its nodes. Attempts to horizontally scale consensus by adding more ordering service nodes eventually lead to throughput degradation [9, 10]. Fortunately, decoupling the ordering and validation steps allows us to mitigate this limitation by devising a scalable communication layer responsible for efficient block dissemination.

Fabric utilizes gossip [7], an epidemic multicast protocol, to address the requirement for efficient ledger replication middleware. In addition, the communication layer needs to provide the capability to synchronize peers that were disconnected for a long time or joined the network late. The ordering service provides a cryptographic signature over the block, thus enabling peers to attest the integrity of disseminated blocks.

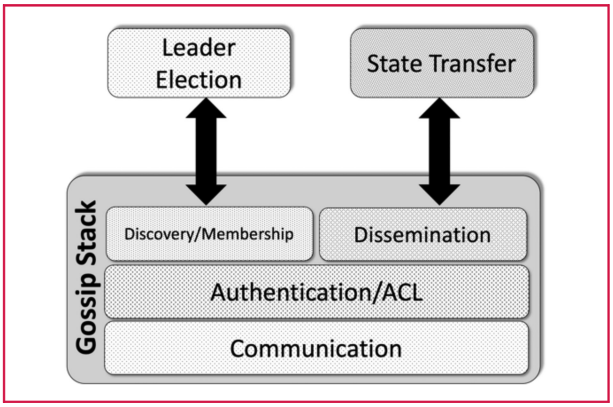

Figure 3

Gossip layer stack.

3.1 Block dissemination

Gossip protocols exchange traditional strong reliability guarantees [11] in favor of a probabilistic approach, leading to greater scalability and fault tolerance. In a "push" gossip protocol, each node selects a random subset of peers from its full membership set and forwards a message to the subset. This mechanism provides a probabilistic guarantee of eventually delivering the message to the entire group of members. In a "pull" gossip protocol, a peer checks with a random subset of peers whether it missed a message, and requests only the messages it missed. The HLF gossip layer uses a combination of push and pull gossip to reliably and efficiently disseminate the transaction blocks ordered by the ordering service. **Figure 3** depicts a high-level architecture of the gossip layer in HLF and its key components.

1) *Communication*: The communication layer for gossip is based on gRPC and utilizes Transport Layer Security (TLS) with mutual authentication, which enables each side of the connection to bind the TLS credentials to the identity of the remote peer.
2) *Authentication/ACL*: It is in charge of authenticating remote peers, validating and storing peer certificates, and enforcing the segregation of information introduced by channels.
3) *Dissemination*: It forwards (pushes) and pulls messages to/from peers according to routing policies (i.e., peer's organization, channels, message type). HLF gossip uses two phases for information dissemination: During push, each peer selects a random set of active neighbors from the membership view and forwards them the message; during pull, each peer periodically probes a set of randomly selected peers and requests missing messages. It has been shown [12, 13] that using both methods in tandem is crucial to optimally utilize the available bandwidth and to ensure that all peers receive all messages with high probability.

4) *Discovery/Membership*: The gossip component maintains an up-to-date membership view of the online peers in the system. All peers independently build a local view from periodically disseminated membership data. Furthermore, a peer can reconnect to the view after a crash or a network outage.
5) *Leader Election*: In order to reduce the load of sending blocks from the ordering nodes to the network, the protocol also elects a leader peer that pulls blocks from the ordering service on their behalf and initiates the gossip distribution. This mechanism is resilient to leader failures.
6) *State Transfer*: A point-to-point replication mechanism that brings new nodes rapidly up to speed by having peers request blocks in batches from peers with higher ledger heights and having those retrieve them directly from the ledger.

3.2 Membership and metadata dissemination

The HLF gossip protocol depends on the ability to randomly select peers from the current membership view; thus, the scalability aspects of the communication layer requires a decentralized membership protocol to be used by epidemic-based algorithms. Essentially, Fabric utilizes a gossip protocol to maintain a membership list in a highly efficient and scalable way similarly to Lpbcast [14] and Newcast [15], achieving strong connectivity properties.

In HLF, peers exchange information about available peers by replicating their "alive" messages, which includes information about peer endpoint (hostport), timestamp, peers identifiers, and peer's incarnation time. In addition, the gossip layer is responsible for disseminating metadata that pertains to the ledger status and the configuration of chaincode and endorsement policies. This includes information, such as ledger height, chaincodes that are active in the channel, and on which peers those chaincodes are installed. This metadata is then leveraged by the discovery service in the peer in order to know which peers can execute given chaincodes.

4 Endorsement policies

The chaincode execution phase is decoupled from the ordering and validation phases by means of using scalable "execute-verify" replication technique [16] adopted to the Byzantine environment. Agreement on execution results is governed by endorsement policies, i.e., every transaction is executed by a subset of peers allowing for parallel execution.

Due to the *permissioned* nature of Fabric, each node in HLF network has an identity that certifies his affiliation to one of the organizations forming the blockchain network. Each identity is associated with a *membership service provider* (MSP)—a modular abstraction that authenticates identities in the system. MSPs reside in all peer and orderer nodes and are used for identity validation, signature verification, and are used in endorsement policy verification.

An endorsement policy in HLF specifies the peers or number of peers that are required to provide an attestation of proper execution of a given chaincode. Endorsement policies are evaluated prior to block commit, during the transaction validation phase. As part of the endorsement policy validation, the signature over the chaincode execution results are verified under the public key of the endorser peer's identity. The endorsement policy is in fact more expressive than just identities—it requires an identity from an organization and a specific role, where roles can be, for example: "Member," "Auditor," etc. The combination of organization and role is called a *Principal*.

Consider an example, as outlined in Figure 1, where the endorsement policy for chaincode *SampleCC* is defined as follows:

$$\textbf{AND}\ (\text{OrgA.Member}, \textbf{OR}\ (\text{OrgB.Member}, \text{OrgC.Member})) \quad (1)$$

while available endorser peers can be: *{peer1.OrgA, peer2.OrgA, peer3.OrgA, peer1.OrgB, peer2.OrgB, peer1.OrgC, peer2.OrgC, peer3.OrgC}* (assuming that all those peers satisfy the "Member" role). Therefore, to satisfy the given endorsement policy, one would have to ask endorsement of a peer from *OrgA* and one peer of either *OrgB* or *OrgC*. Clearly, the endorsement policy could be satisfied in more than one way.

Let us describe how endorsement policies are represented and validated. An endorsement policy is composed of the following two objects.

1) An array of principals: A principal is a predicate over an identity, meaning that every identity either satisfies or does not satisfy a given principal (e.g., by having a certain role).
2) A tree that represents principal sets that correspond the endorsement policy. At least one of the sets should be satisfied entirely in order for the endorsement policy to be deemed satisfied. The tree has the following two types of vertices.
   a) *NoutOf*: Inner vertices (nonleaves) are quantifiers, i.e., specifies how many of its direct descendants should be satisfied to consider vertex itself as satisfied.
   b) *SignedBy*: Leaf vertices are pointers to the array of principals, which give the endorsement policy expression power of security roles and conditions over identities.

Even though quantifiers can denote any number from 1 to the out degree of a vertex, the most common use cases are **1OutOf** and **ALLoutOf,** which represent logical gates **or** and **and**, respectively. Consider example 1 of endorsement

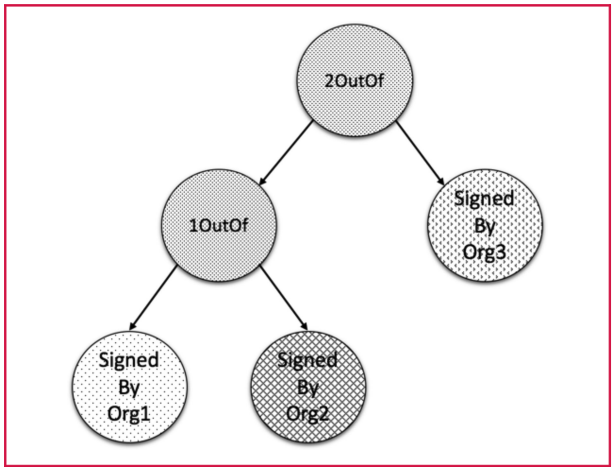

Figure 4

Endorsement tree corresponding to the example shown in Figure 1, for chaincode **SampleCC.**

policy; the corresponding endorsement tree is depicted in **Figure 4**. Correspondingly, the possible principal sets that will satisfy the endorsement policy would be *{OrgA,OrgB}*, *{OrgA,OrgC}*, and *{OrgA,OrgB,OrgC}*[1]; or basically each combination of members from **OrgA** and of **OrgB** or **OrgC**.

More formally, to evaluate the endorsement policy for a given set of peer identities, the endorsement tree is traversed recursively to find the first combination of principals that is satisfied with the given set of identities. The discovery service computes a descriptor that enables to compute all possible combinations of peers, such that endorsements collected from every combination satisfy the endorsement policy.

4.1 Computation of principals sets

In the discovery service, an endorsement policy is evaluated to produce combinations of principals such that every combination satisfies the endorsement policy on its own. To compute the combinations of principals, the endorsement policy tree is traversed, and subtrees are computed such that the leaf level in every such subtree is a combination of principals that satisfies the policy. This is done by computing for each inner vertex with an *NoutOf* quantifier of *n*, all permutations of its descendants of size *n*. Afterward, the endorsement tree is traversed in BFS, and for each inner vertex, the subtree is duplicated to accommodate all permutations, until all the vertices are visited.

Note that for every such principal set that satisfies the endorsement policy, there might be pluralities of principals required. For instance—an endorsement policy might require multiple signatures of different peers from the same organization.

In order to represent such a principal set in a space-efficient manner, the principal set is saved as a mapping from principals to their pluralities. Each such mapping is a combination of principals and is called a layout. For example, a single **layout** would look like this, *{'OrgA.Member': 2, 'OrgB.Auditor': 1},* to indicate that two "**Member**" peers from "**OrgA**" and one "**Auditor**" peer from "**OrgB**" would satisfy the endorsement policy.

Going back to the example shown in Figure 4, the endorsement descriptor corresponding to the endorsement policy represented by the example's chaincode **SampleCC** will be structured as follows.

Listing 1 Endorsement Descriptor

```
Layouts: [
{
    "OrgA": 1,
    "OrgB": 1,
},
{
    "OrgA": 1,
    "OrgC": 1,
}],
EndorsersByGroups: {
"OrgA":  [peer1.OrgA,peer2.OrgA,peer3.
OrgA],
"OrgB": [peer1.OrgB,peer2.OrgB],
"OrgC":  [peer1.OrgC,peer2.OrgC,peer3.
OrgC]
}
```

Given a layout, there are multiple ways to select peers such that they satisfy the principals in the layout. Therefore, the discovery service also computes a mapping from principals to peers (the `EndorsersByGroups` mapping in Listing 1). This computation is achieved by utilizing a bipartite graph matching algorithm [17], in which the left side of the bipartite graph represents principals, the right side represents peer identities, and an edge between two vertices exists iff the peer identity satisfies the principal. If all principal vertices are covered in the matching, it means there is a valid assignment of every principal to a single peer.

Note: This selection does not take into account network-level information that would result in a more appropriate selection of peers, such as preferring peers with higher ledger heights than other peers or excluding peers that are found to be offline by the client, etc., and should be handled by the client SDK.

5 Service discovery

In order to execute chaincode on peers, submit transactions to orderers, and to be updated about the status of

[1]This set is redundant. Usually minimal sets are supplied; see next section. For brevity, we assume all peers have the same role.

transactions, applications connect to an API exposed by an SDK, as outlined in Section 2.2.

However, the SDK needs a lot of information in order to allow applications to connect to the relevant network nodes. It needs to know the enrollment CA and TLS CA certificates of the orderers and peers on the channel, as well as their IP addresses and port numbers. In addition, it must know the relevant endorsement policies that are coupled with the chaincode that the peers have installed on them. This is necessary so that the application knows to which peers to send chaincode proposals.

In early versions of HLF (prior to v1.2), this information was statically encoded. However, that implementation was not dynamically reactive to network changes (such as the addition of peers who have installed the relevant chaincode, or peers that are temporarily offline). Static configurations also do not allow applications to react to changes of the endorsement policy itself (as might happen when a new organization joins a channel). Furthermore, the client application had no way of knowing which peers have updated ledgers and which do not, so it might submit proposals to peers whose ledger data are not in sync with the rest of the network, resulting in transaction being invalidated upon commit. That was a waste of both time and resources.

The *discovery service* improves this process by having the peers compute the needed information dynamically and present it to the SDK in a consumable manner.

5.1 How service discovery works in Fabric

The application is bootstrapped knowing about a group of peers that are trusted by the application developer/administrator to provide authentic responses to discovery queries. A good candidate peer to be used by the client application is one that is in the same organization.

The application issues a configuration query to the discovery service and obtains all the static information it would have otherwise needed to communicate with the rest of the nodes of the network. This information can be refreshed at any point by sending a subsequent query to the discovery service of a peer.

The service runs on peers—not on the application—and uses the network metadata information maintained by the gossip (see Section 3) to render the list of peers that are online. It also fetches information, such as relevant endorsement policies, from the peer's state database.

With service discovery, applications no longer need to specify from which peers they need endorsements. The SDK can simply send a query to the discovery service asking which peers are needed given a channel and a chaincode ID.

The discovery service can respond to the following queries.

1) *Configuration query*—returns the configuration required for initialization of the CA certificates of all organizations in the channel along with the orderer endpoints of the channel.
2) *Peer membership query*—returns the peers that have joined the channel. Additional metadata, such as the chaincodes that are installed, the certificates of the peers, and the ledger height, is included in the information.
3) *Endorsement query*—returns an *endorsement descriptor* for given chaincode(s). The descriptor allows easy selection of some set of peers such that if endorsements are obtained from the set, the endorsement policy would be satisfied. The same metadata on peers that is returned in the membership query is also included in the results.
4) *Local peer membership query*—returns channel-oblivious information known to the peer, i.e., all peers it knows about, regardless of channels.

5.2 Chaincode to chaincode invocation and endorsement queries

A chaincode may also invoke another chaincode during its execution. In such a scenario, the resulting transaction can affect several namespaces of the world state, and not just the namespace of the target chaincode to which the client sent the transaction proposal. At validation time, such a transaction is valid only if the endorsements satisfy the endorsement policies of all the chaincodes denoted in the transaction, and not just the target chaincode's endorsement policy. The discovery service supports these types of scenarios by computing principal sets of all chaincodes in the invocation chain of an endorsement query and by the following.

1) Merging principal sets that cover other principal sets.
2) Unifying principal sets that are disjoint between the various chaincodes in the query's invocation chain, such that every principal set satisfies all endorsement policies.

5.3 Private data collections and endorsement queries

HLF also possesses a mechanism to share data among a subset of the channel members—called "private data collection" (or collection). This is done by storing in the transaction simulation results the hashes (using a cryptographic hash function, such as SHA256) of the data itself and disseminating the hash pre-images only among the peers that are members of the collection. This adds another hurdle to the endorser selection of the client—a peer that is not part of a collection cannot simulate transactions that use keys that are known only to peers that are members of the collection. Furthermore, the client's input to the chaincode might contain sensitive information

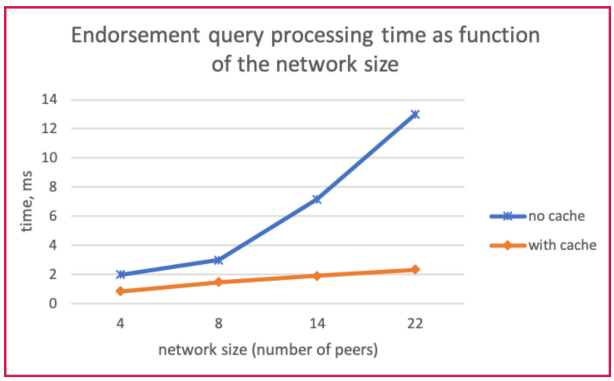

Figure 5

Time taken to process endorsement query via *service discovery* as a function of network size.

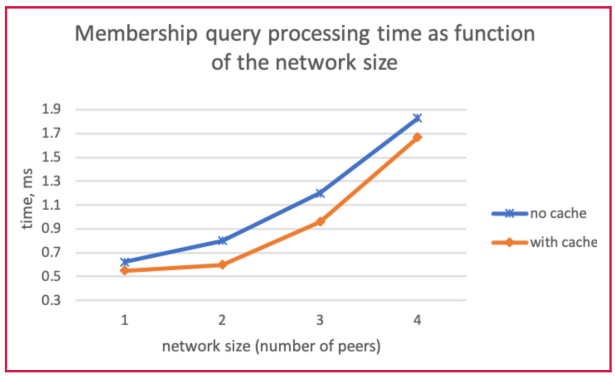

Figure 6

Time taken to process membership query via *service discovery* as a function of network size.

that should be hidden from peers that are not members in the collection. Therefore, the client needs to send proposals to peers that are members of the collection and avoid sending to those which are not. The discovery service addresses this requirement by having the client's endorsement queries specify collections per chaincode and returning to the client a descriptor that contains only peers that are part of the collection.

6 Evaluation

We evaluated the performance handling of requests that contain either an endorsement query or a membership query. We use a single x86 VM with 4 CPU 2.7 GHz and 10 GB of RAM and measure the query processing time as a function of network size. The query is executed against a single peer; the rest of the peers in the network are idle and are introduced only to vary the difficulty of processing the query. It is, therefore, valid to use a single VM, as only a single peer consumes compute resources at the time of the query.

The time measured was the time it took for the request to be processed and does not include the round-trip time between the client and the peer. The benchmarking entailed invoking an endorsement or a membership query on the same peer twice on network configurations that differ in the number of peers in the channel. **Figures 5** and **6** present the time it takes to process an endorsement query and a membership query, respectively, as a function of network size. Each figure presents two curves: one with and one without the use of a cache.

Processing an endorsement query amounts to principal matching to peers, which involves the assessment of whether a given peer identity satisfies a given principal (see Section 4.1). This computation is done by the MSP peer component, and involves Elliptic Curve Digital Signature Algorithm (ECDSA) signature verification. Signature verification was found to be the major contributor to request handling duration. Therefore (as is clearly shown in Figure 5, no-cache), the query processing time increases with the number of different peers.

Fortunately, the MSP component has a caching layer that significantly reduces the processing time of subsequent queries, as long as peers are not replaced. As shown in Figure 5, the caching of whether peer identities satisfy principals significantly shortens the computation time in a network with 22 peers from 13 ms to a mere 2.34 ms ($\sim$5.5 $\times$ speedup).

Membership queries are much simpler than endorsement queries, as they do not require principal matching to peers. Therefore, the effect of caching on these queries, as shown in Figure 6, is less dramatic and improved only 10% to 20% of the processing time.

7 Conclusion

HLF was the first blockchain platform to employ the *execute-order-validate* pattern. This innovation decouples the execution and endorsement of a transaction from its total ordering and commitment to the ledger and opens the door for parallel execution of independent transactions. This architecture also addresses the problem of nondeterministic chaincode execution since such transactions are filtered away either by the client that collects endorsements or by the validation phase that enforces valid endorsements.

However, these advantages come at the cost of added complexity. The transaction flow in HLF is more complicated compared to Bitcoin or Ethereum, for example, and the onus of coordinating this flow falls on the client. The client has to communicate with, and be aware of, multiple entities: the chaincode location, the endorsement policy, the endorsing peers, and the ordering service. All of these entities are subject to change during the life-cycle of the platform: Peers and organizations may come and go,

chaincode may be upgraded, and endorsement policies may be updated. The fact that HLF is a permissioned ledger means that the configuration of the client involves the intricacies of authentication, authorization, and access control, which can be very technical and error-prone.

The *service discovery* component helps the client application code deal with the complexity and dynamic nature of the platform. It hides a lot of the complexity involved in configuring the client by providing, via the client SDK, a set of APIs that ease and automate the job of configuring the client. By allowing the client to easily and automatically adapt to changes in the platform, the reliability and availability of the blockchain application is significantly increased. The service discovery component also simplifies the task of the application developer, making the task of writing robust application code much easier.

References

1. S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008.
2. V. Buterin, "A next-generation smart contract and decentralized application platform," Ethereum white paper, 2014.
3. F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.
4. E. Androulaki, A. Barger, V. Bortnikov, et al., "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, Porto, Portugal, Apr. 2018, paper 30. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538
5. N. Budhiraja, K. Marzullo, F. B. Schneider, et al., "The primary-backup approach," *Distrib. Syst.*, vol. 2, pp. 199–216, 1993.
6. B. Charron-Bost, F. Pedone, and A. Schiper, *Replication* (Lecture Notes in Computer Science Series), vol. 5959. Berlin, Germany: Springer, 2010, pp. 19–40.
7. A. Barger, Y. Manevich, B. Mandler, et al., "Scalable communication middleware for permissioned distributed ledgers," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, paper 23.
8. C. Cachin, S. Schubert, and M. Vukolić, "Non-determinism in Byzantine fault-tolerant replication," in *Proc. 20th Int. Conf. Principles Distrib. Syst. Leibniz Int. Proc. Inf.*, 2016, pp. 24:1–24:16.
9. K. Croman, C. Decker, I. Eyal, et al., "On scaling decentralized blockchains," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2016, pp. 106–125.
10. M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Proc. Int. Workshop Open Problems Netw. Secur.*, 2015, pp. 112–125.
11. K. Birman and T. Joseph, Exploiting virtual synchrony in distributed systems. *ACM Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, 1987.
12. A. Demers, D. Greene, C. Hauser, et al., "Epidemic algorithms for replicated database maintenance," in *Proc. 6th Annu. ACM Symp. Principles Distrib. Comput.*, 1987, pp. 1–12.
13. R. Karp, C. Schindelhauer, S. Shenker, et al., "Randomized rumor spreading," in *Proc. Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, pp. 565–574.
14. P. T. Eugster, R. Guerraoui, S. B. Handurukande, et al., "Lightweight probabilistic broadcast," *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 341–374, 2003.
15. M. Jelasity, W. Kowalczyk, and M. v. Steen, "Newscast computing," Dept. Comput. Sci., Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, Internal Rep. IR-CS-006, 2012.
16. M. Kapritsos, Y. Wang, V. Quema, et al., "All about eve: Execute-verify replication for multi-core servers," in *Proc. Symp. Oper. Syst. Des. Implementation*, 2012, vol. 12, pp. 237–250.
17. J. E. Hopcroftand and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, no. 4, pp. 225–231, 1973.

Received July 18, 2018; accepted for publication February 1, 2019

Yacov Manevich *IBM Research, Haifa 31905, Israel (yacovm@il.ibm.com).* Mr. Manevich received a B.Sc. degree in computer science from Technion—Israel Institute of Technology, Haifa, Israel, in 2016. He is currently a Research Staff Member with IBM Research - Haifa, where he is working in the area of Blockchain, and is a maintainer of the Hyperledger Fabric core.

Artem Barger *IBM Research, Haifa 31905, Israel (bartem@il.ibm.com).* Mr. Barger received a B.Sc. degree in computer science from Technion—Israel Institute of Technology, Haifa, Israel, in 2010, and the M.Sc. degree in computer science from University of Haifa, Haifa. He is currently a Research Staff Member with IBM Research - Haifa, where he is working in the areas of distributed computing, recently focusing on Blockchain domain developing and contributing to the Hyperledger Fabric open-source project.

Yoav Tock *IBM Research, Haifa 31905, Israel (tock@il.ibm.com).* Dr. Tock received B.Sc. (*cum laude*), M.Sc., and Ph.D. degrees from the Electrical Engineering faculty, Technion—Israel Institute of Technology, Haifa, Israel, in 1994, 1999, and 2003, respectively. He is currently a research staff member with the IBM Research - Haifa, where he is working in the areas of scalable and highly available distributed middleware.