CAPER: A Cross-Application Permissioned Blockchain

Mohammad Javad Amiri Divyakant Agrawal Amr El Abbadi
Department of Computer Science, University of California Santa Barbara
Santa Barbara, California
{amiri, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Despite recent intensive research, existing blockchain systems do not adequately address all the characteristics of distributed applications. In particular, distributed applications collaborate with each other following service level agreements (SLAs) to provide different services. While collaboration between applications, e.g., cross-application transactions, should be *visible* to all applications, the internal data of each application, e.g, internal transactions, might be *confidential*. In this paper, we introduce *CAPER*, a permissioned blockchain system to support both internal and cross-application transactions of collaborating distributed applications. In CAPER, the blockchain ledger is formed as a *directed acyclic graph* where each application accesses and maintains only its own *view* of the ledger including its internal and all cross-application transactions. CAPER also introduces three consensus protocols to globally order cross-application transactions between applications with different internal consensus protocols. The experimental results reveal the efficiency of CAPER in terms of performance and scalability.

PVLDB Reference Format:
Mohammad Javad Amiri, Divyakant Agrawal, Amr El Abbadi. CAPER: A Cross-Application Permissioned Blockchain. *PVLDB*, 12(11): 1385-1398, 2019.
DOI: https://doi.org/10.14778/3342263.3342275

1. INTRODUCTION

Blockchain, originally devised for the Bitcoin cryptocurrency [40], is a distributed data structure for recording transactions maintained by nodes without a central authority [17]. In a blockchain, nodes agree on their shared states across a large network of *untrusted* participants. The unique features of blockchain such as transparency, provenance, fault tolerance, and authenticity are used by many systems to deploy a wide range of distributed applications such as supply chain management [31] and healthcare [12] in a permissioned settings. Unlike *permissionless* settings, e.g., Bitcoin [40], where the network is public, and anyone can participate without a specific identity, a *permissioned* blockchain consists of a set of known, identified nodes that might not fully trust each other.

To support distributed applications, different characteristics of such applications need to be addressed by permissioned blockchain systems. Distributed applications require high performance in terms of throughput and latency, e.g., a financial application needs to process tens of thousands of requests every second with very low latency. Distributed applications also collaborate with each other to provide different services. Collaborations are defined in service level agreements (SLAs) which are agreed upon by all involved applications. SLAs can be written as self executing computer programs, called *smart contracts* [3]. The collaboration is realized by means of cross-application transactions that are *visible* to every application. During the execution of cross-application transactions, agreement on the shared state of the collaborating applications is needed without trusting a central authority or any particular participant. While cross-application collaborations and the involved data are *visible* to every application, the internal data of each application, i.e., the application logic, internal transactions, and their data, might be *confidential*. Hence, it is desirable to restrict access to such data. Although cryptographic techniques can be used to achieve confidentiality, the high overhead of such techniques makes them impractical [10].

Existing permissioned blockchains mostly suffer from performance issues in terms of throughput and latency because of the sequential execution of transactions on all nodes, and also confidentiality issues since a single blockchain ledger with all transactions is maintained at every node.

Hyperledger Fabric [10], on the other hand, improves performance by executing the transactions of different applications that are deployed on the same channel in parallel and addresses the confidentiality leaks by restricting access to the blockchain state (using private data collections) and *smart contracts* which include the application logic. However, the blockchain ledger is still maintained by every node.

To provide confidentiality, different applications could have independent disjoint blockchains. However, to support collaboration between applications on distinct blockchains, techniques such as atomic cross-chain swap [27] and Interledger protocol [49] are needed to exchange (transfer) assets or information between the blockchains (*Interoperability*). Such techniques are often costly, complex, and mainly designed for permissionless blockchains. Techniques that support collaborating applications on a single blockchain either do not

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342275

support internal transactions of applications [51] (results in data integration issues), or suffer from confidentiality issues since the entire ledger is visible to all applications, e.g., single-channel Fabric [10], or require a trusted channel among participants, e.g., multi-channel Fabric [11].

In this paper, we present *CAPER*: a permissioned blockchain system that supports both internal and cross-application transactions of collaborating distributed applications. In CAPER, each application orders and executes its internal transactions locally while cross-application transactions are public and visible to every node. In addition, the blockchain ledger of CAPER is a directed acyclic graph that includes the internal transactions of every application and all cross-application transactions. Nonetheless, for the sake of confidentiality, the blockchain ledger is *not maintained* by any node. In fact, each application maintains its own *local view* of the ledger including its internal and all cross-application transactions. Since ordering cross-application transactions requires *global* agreement among all applications, CAPER introduces different consensus protocols to globally order cross-application transactions.

Lack of trust is an important problem in collaboration between applications. Lack of trust has two main origins: first, a *physical node* (server) might fail, thus behave maliciously, and second, an *application* could behave maliciously in the communication with other applications for its benefits. To address both types of behavior, CAPER distinguishes between trust at the node level and trust at the application level, e.g., while the nodes of an application might behave non-maliciously within the application, the application (as a collection of nodes) might still behave maliciously in communication with other applications.

A key objective of this paper is to demonstrate how internal and cross-application transactions of a set of collaborating distributed applications which do not trust each other can be processed efficiently by a blockchain system while both confidentiality constraints of internal transactions and visibility constraints of cross-application transactions are met. The contributions of this paper are three-fold:

- Introducing *Blockchain views* where each application maintains *only* its own *view* of the ledger including its internal and all cross-application transactions.
- CAPER, a permissioned blockchain that supports collaborating distributed applications. CAPER supports both internal and cross-application transactions.
- Three different consensus protocols for globally ordering cross-application transactions among applications with different local consensus protocols.

The rest of this paper is organized as follows. Section 2 briefly describes the limitations of current blockchain systems and motivates the problem. The CAPER model and architecture are introduced in Section 3 and Section 4. Section 5 and Section 6 present consensus in CAPER. Section 7 presents a performance evaluation of CAPER. Section 8 discusses related work, and Section 9 concludes the paper.

2. BACKGROUND AND MOTIVATION

A blockchain is a distributed data structure for recording transactions, maintained by nodes without a central authority [17]. Existing blockchain systems can be divided into two main categories: permissionless blockchains, e.g, Bitcoin [40], and permissioned blockchains, e.g., Tendermint [32].

2.1 Limitations of Permissioned Blockchain

Blockchains have unique features, such as transparency, provenance, fault tolerance, and authenticity, which appeal to a wide range of distributed applications, e.g., supply chain management [31] and healthcare [12]. However blockchain systems suffer from some limitations. In the following, we mainly focus on permissioned blockchains and discuss three of their significant limitations: poor *performance*, lack of *confidentiality*, and inefficient *cross-application transaction* support. Note that some of these limitations exist in permissionless blockchain systems as well.

Existing blockchains mostly utilize an order-execute architecture where nodes agree on a total order of the blocks of transactions using a consensus protocol and then the transactions are executed in the same order on all nodes sequentially. The sequential execution of transactions, however, reduces the blockchain performance in terms of throughput and latency. To address this issue, Hyperledger Fabric [10] presents a new architecture for permissioned blockchains by switching the order of the execution and ordering phases. In Fabric, the transactions of different applications are first executed in parallel and then an ordering service establishes consensus on a total order of requests.

While Fabric reduces the latency of the system by executing transactions in parallel, the sequential construction of blocks could still be a performance bottleneck when the system is deployed on a single channel or if channels use the same ordering service. In Bitcoin [40], (single-channel) Fabric [10], and many other permissioned and permissionless blockchains, blocks cannot be constructed simultaneously and only a single chain is allowed on the entire network. In contrast to this linear structure, some other blockchains, e.g., Iota [44], and Hashgraph [14], are structured as a directed acyclic graph (DAG). The DAG structure can increase the throughput of transactions by exploiting parallel construction resulting in the parallel execution of transactions in different blocks. In such a structure, the blocks (transactions) that are independent of each other can be constructed and added to the ledger simultaneously.

Batching transactions into blocks is another reason for the low performance of blockchains. Blocks were originally created, first, to amortize the cost of cryptography, e.g., solving proof-of-work, and second, to make data transfers more efficient in a large geo-distributed setting. However, StreamChain [28] demonstrates that in permissioned blockchains, since proof-of-work is not required, batching transactions into blocks decreases performance.

Besides the performance issues, confidentiality of data is required in many permissioned blockchains. A blockchain might need to restrict access to smart contracts (which include the logic of applications), blockchain ledger, and transaction data. While cryptographic techniques can be used to achieve confidentiality, the considerable overhead of such techniques makes them impractical [10]. Fabric ensures the confidentiality of data using Private Data Collections [7]. Private Data Collections manage confidential data that two or more applications on a single channel want to keep private from other applications on that channel. The confidentiality of smart contracts in Fabric is also ensured by storing

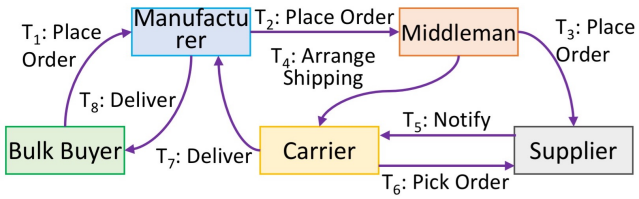


Figure 1: A Supply Chain Scenario

the smart contracts of different applications on different sets of nodes, called endorsers. The endorsers of an application are responsible for executing the transactions of the application independent of the other applications. However, the blockchain ledger is still maintained by the agents of every application within a channel.

In addition to performance and confidentiality issues, many applications need to collaborate with each other to provide different services. Distributed applications are often designed and implemented in different blockchain systems, each of which processes and stores data independently [29]. In this case, inter-application collaboration could be performed as an atomic cross-chain swap [27] or using the Interledger protocol [49] where two applications (parties) exchange (transfer) assets or information across their blockchains. Atomic cross-chain transactions are also addressed between permissioned blockchains (channels) in Fabric by either assuming the existence of a trusted channel among the participants or using an atomic commit protocol [11]. In general, supporting cross-application transactions is an expensive and challenging task. Nonetheless, for collaborating applications in a single permissioned blockchain, since the participants of cross-application transactions are known beforehand and the transactions follow service level agreements, we might be able to find a solution that preserves both confidentiality and performance.

2.2 A Motivating Example: Supply Chain

We now consider collaborative workflows as a use-case to illustrate the aforementioned limitations. In a collaborative workflow, different parties need to communicate across organizations to provide services. However, the lack of trust between parties is problematic.

Figure 1 shows a Supply Chain scenario (reported in [51]). The workflow involves five participants (applications): Supplier, Manufacturer, Bulk Buyer, Carrier, and Middleman where each of the participants might have multiple trusted or untrusted nodes that perform different internal tasks (transactions). For example, the production process in the Manufacturer involves Financial, Marketing, and Purchasing departments and includes different internal tasks such as assembly, painting, drying, testing, and packaging. Participants also need to communicate with each other to provide different services. The Bulk Buyer communicates with the Manufacturer to place an order, the Manufacturer places an order for materials via a Middleman, the Middleman forwards the order to a Supplier and arranges transportation by a Carrier. Once the materials are acquired, the Supplier informs the Carrier and the Carrier picks them up and delivers them to the Manufacturer. As soon as the goods become available, the Manufacturer delivers them to the Bulk Buyer. These collaborations are defined in service level agreements which are agreed upon by all participants.

Now the Manufacturer might receive the materials later than agreed upon or might receive something different from what they agreed on. In this case, the Supplier might argue that this is exactly what was ordered by the Middleman while the Middleman would blame the Supplier. The situation is complicated for the Carrier since the Manufacturer might refuse to accept the delivery. The Carrier is now eligible for compensation from either the Supplier or the Middleman depending on who is responsible for the fault.

To tackle such an issue, *permissioned* blockchain systems can be used among all the different participants to ensure agreement on the shared state of the collaborating parties without trusting a central authority or any particular participant [51]. The blockchain basically *monitors* the execution of the collaborative process and *checks conformance* between the process execution and SLAs. Any blockchain-based solution has to address the following concerns.

First, the blockchain system should support both cross-application and internal transactions. For example, in the Supply Chain scenario, in addition to cross-application transactions, the Manufacturer might want to use the blockchain for its internal transactions, e.g., calculating materials demand or testing the product, to benefit from the the unique features of blockchain. Second, in contrast to the cross-application transactions which are public and can be accessed by all participants, the internal transactions of each application and their data should only be accessed by the nodes of the application to preserve confidentiality, e.g., the internal transactions of the Manufacturer show its internal process for producing a product which the Manufacturer might intend to keep as a secret. Third, the solution has to address the performance aspect as well.

One possible solution is to implement all applications within a single blockchain where all the transactions are maintained in a single blockchain ledger which is replicated among all the nodes in the blockchain. This solution handles the cross-application transactions efficiently because every node accesses all the data. However, since the ledger is replicated among all the nodes and every transaction is visible to all applications, the confidentiality of data is not preserved.

Another solution is to implement each application on a separate blockchain. In that way, participants can perform their internal transactions in parallel resulting in higher performance and since their data is maintained on different blockchains, the confidentiality of data is preserved. To perform communication between different applications, one approach is to use cross-chain swap operation, which, as discussed earlier, is expensive. An alternative approach is to use a new blockchain to maintain public transactions. However, since public transactions use data which is provided by internal transactions, e.g., to place an order, the Manufacturer needs to calculate demand internally, and these two types of transactions are stored in different blockchains, *data integration* becomes an expensive and challenging task.

In this paper, we present a new approach that not only addresses the performance and confidentiality issues, but also handles cross-application transactions efficiently.

3. THE CAPER MODEL

In this section, the CAPER model is introduced. We first present distributed applications and the blockchain ledger, and then show how the distributed applications are deployed in the blockchain.

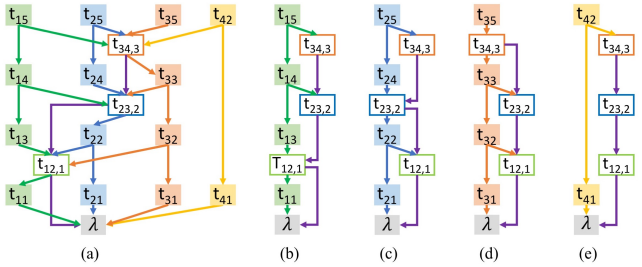


Figure 2: (a): A blockchain ledger consisting of Four applications, (b), (c), (d), and (e): The views of the blockchain from different applications

3.1 Distributed Applications

CAPER is a blockchain system designed to support a set of collaborating distributed applications which might not trust each other. Each application maintains two sets of *private* and *public* records. The private records of an application are accessible only to the application whereas the public records are replicated on all applications.

CAPER supports internal and cross-application transactions. *Internal* transactions are performed within an application following the logic of the application, e.g., in the Supply Chain scenario, the Manufacturer calculates materials demand internally. Internal transactions of an application can read and write its private records, however they can only read (and not write) the public records. *Cross-application (public)* transactions, on the other hand, involve multiple applications and are visible to all applications, e.g., the Manufacturer places an order for materials via a Middleman. Cross-application transactions follow the service level agreements (SLAs) between the involved applications. SLAs present the flow of communication between the applications and indicate different aspects of the services, e.g., quality, availability, and responsibilities, that should be provided by different applications. For example, in the Supply Chain scenario, the Carrier is responsible for delivering requested materials to the Manufacturer in two business days from the date it is informed by the Supplier. Public records can only be updated via Cross-application transactions.

3.2 Blockchain Ledger

The *blockchain ledger* is an append-only data structure recording transactions in the form of a hash chain where each block contains a batch of transactions. In a permissioned blockchain, as discussed in Section 2, batching transactions into blocks decreases performance (Since proof-of-work is not required [28]). Thus, in CAPER, each block consists of a single transaction. The blockchain ledger in CAPER consists of both internal transactions of all applications and all cross-application transactions in the system. To support both types of transactions, we generalize the notion of a blockchain ledger from a linear chain to a *directed acyclic graph (DAG)* where *nodes* of the graph are transactions and *edges* enforce the order of transactions.

Within an application, since transactions have access to the same datastore, a total order between the transactions that are initiated by the application is enforced to ensure consistency. To present the total order of transactions in the blockchain ledger, transactions are *chained* together, i.e., each transaction includes the cryptographic hash of the previous transaction. In addition, since cross-application transactions update data which is replicated on all the applications, to ensure consistency, cross-application transactions are totally ordered as well. Furthermore, internal transactions of applications might use data that is provided by cross-application transactions, e.g., the Manufacturer calculates materials demand based on the place-order cross-application transaction of the Bulk Buyer. To show such data dependencies, an internal transaction includes the cryptographic hash of a cross-application transaction in the ledger.

In summary, the blockchain ledger has three properties: (1) There is a total order between all transactions (internal as well as cross-application) that are initiated by an application, (2) There is a total order between cross-application transactions, and (3) An internal transaction of an application might include the cryptographic hash of a cross-application transaction (that is initiated by another application).

In addition to internal and cross-application transactions, a unique initialization transaction (block), called *genesis* transaction, is considered for the blockchain. Function $H(.)$ also denotes the cryptographic hash function. For simplicity, to show that transaction t includes $H(t')$ (i.e. t is ordered immediately after t' as explained in properties 1 and 2 or has data dependency to t' as explained in property 3) we include an *edge* from t to t' in the DAG representation of the blockchain ledger.

Fig. 2(a) shows a CAPER blockchain ledger consisting of four applications α_1, α_2, α_3, and α_4. In this figure, λ is the genesis transaction. Internal and cross-application transactions of each application are also specified. For example, t_{11}, t_{13}, t_{14}, and t_{15} are the internal transactions of application α_1, and $t_{12,1}$, $t_{23,2}$, and $t_{34,3}$ are the cross-application transactions initiated by α_1, α_2, and α_3 respectively. Note that each cross-application transaction is labeled with $t_{i,j}$ where i indicates the order of the transaction among the transactions that are initiated by its initiator application and j presents the order of the transaction among all cross-application transactions. As can be seen, transactions (both internal and cross-application) that are initiated by an application are chained together (property 1), e.g. t_{31}, t_{32}, t_{33}, $t_{34,3}$, and t_{35}. In addition, cross-application transactions are chained together (property 2), i.e., $t_{12,1}$, $t_{23,2}$, and $t_{34,3}$. Finally, an internal transaction might include the hash of a cross-application transaction that shows the data dependency of the internal transaction to the cross-application transaction (property 3), e.g., internal transactions t_{22} of α_2 has edge to cross-application transaction $t_{12,1}$.

Note that, since an edge from transaction t to transaction t' indicates that t occurs after t' (t includes the hash of t', thus t' has to be appended to the ledger earlier), it is easy to show that the resulting graph is acyclic.

In contrast to the cross-application transactions that are visible to and maintained by all applications, the internal transactions of an application present confidential data about the application, e.g., its business logic. For example, in the Supply Chain scenario, the internal transactions of the Manufacturer show its internal process for producing a product which the Manufacturer might intend to keep as a secret. The presented blockchain ledger, however, is at odds with confidentiality because every application has access to every transaction. For the sake of confidentiality, we want to prohibit an application from observing the internal transactions

of other applications. To achieve this, in CAPER, the entire blockchain ledger is *not maintained* by any application. In fact, each application only maintains its own *view* of the blockchain ledger that includes its internal transactions and all the cross-application transactions. The blockchain ledger is indeed the union of all these physical views.

Fig. 2(b)-(e) show the views of the blockchain ledger for applications α_1, α_2, α_3, and α_4 respectively where each application maintains only the part of the ledger consisting of its internal and all the cross-application transactions.

3.3 Application Deployment on a Blockchain

Each application in CAPER, in addition to the datastore and its view of the blockchain ledger, maintains a "private smart contract" to implement the application logic, and a "public smart contract" to implement the logic of cross-application transactions.

A *smart contract*, as exemplified by Ethereum [3], is a computer program that self-executes once it is established and deployed. Smart contracts are similar to *database triggers* where the logic of the contract is triggered to be executed once some conditions or terms are met. It has the advantages of being updated in real-time, ensuring accurate execution, and requiring little human intervention. A smart contract can also handle automatic conditional payments from escrow. When a payment function is triggered, the smart contract automatically checks the defined conditions, and transfers the money according to the defined rules [51].

Each application in CAPER has its own *private smart contract* which includes the logic of the internal transactions. In addition, a *public smart contract* is written to include the logic of cross-application transactions which is determined by the service level agreements between the applications. As discussed in Section 2, to execute a collaborative process, participants agree on SLAs which are then written in the public smart contracts. The public smart contract runs on every application to check if the cross-application transactions are *conforming* to the SLAs, and enforce the conditions defined in the cross-application transactions. In contrast to most existing blockchains where every smart contract runs on all nodes, which is at odds with confidentiality, in CAPER, each private smart contract runs only on its application. To support cross-application transactions, however, the public smart contract runs on every application. Furthermore, both private and public smart contracts can be written in domain-specific languages, e.g., Solidity, to ensure the deterministic execution of transactions.

In the Supply Chain scenario in Figure 1, each of the five involving applications, i.e., Supplier, Manufacturer, Bulk Buyer, Carrier, and Middleman, executes its internal transactions following the application logic, which is implemented in its private smart contract. Once a cross-application transaction is requested, e.g., the Bulk Buyer places an order with the Manufacturer, every application executes and appends the transaction to its view of the ledger. To execute the cross-application transactions, the public smart contract is used which includes the SLAs (that are agreed upon by all applications). Hence, if the requested transaction does not conform to the SLAs, it will be detected. For each cross-application transaction the SLA defines several conditions to check. SLA also includes actions for non-conforming transactions, e.g., if the Carrier delivers the materials later than agreed upon, it will be penalized as specified in the SLA or if the delivered materials are something different from what they agreed on, the transaction will be aborted.

4. THE CAPER ARCHITECTURE

CAPER consists of a set of nodes in an asynchronous distributed system where each application runs on a (non-empty) disjoint subset of nodes called the *agents* of the application. We use N and A to denote the set of nodes and applications. In addition, N_α indicates the set of agents of application $\alpha \in A$ where for each pair of applications α_1 and α_2 in A, $N_{\alpha_1} \cap N_{\alpha_2} = \varnothing$.

Nodes are connected by point-to-point bi-directional communication channels. Network channels are pairwise authenticated, which guarantees that a malicious node cannot forge a message from a correct node, i.e., if node i receives a message m in the incoming link from node j, then node j must have sent message m to i beforehand.

Furthermore, messages may contain public-key signatures and message digests [19]. A *message digest* is a numeric representation of the contents of a message produced by collision-resistant hash functions. Message digests are used to detect changes and alterations to any part of the message. We denote a message m signed by replica r as $\langle m \rangle_{\sigma_r}$ and the digest of a message m by $D(m)$. For signature verification, we assume that all machines have access to the public keys of all other machines.

Nodes in CAPER might crash, behave maliciously, or be reliable. In addition, we assume that applications do not trust each other, thus we model application failures as Byzantine failures. In fact, we define two levels of behavior in the system. First, at the node level, each agent might be a crash-only, a Byzantine, or a reliable node. In a crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart. Whereas, in a Byzantine failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior. A reliable node, on the other hand, never fails. Second, at the application level, an application (as a group of agents) might behave maliciously. Note that these two levels of behavior are independent of each other. Thus, even if the agents of an application are crash-only nodes, the application might still behave maliciously.

Ordering the transactions within each application needs consensus among the agents of the application. To establish consensus, asynchronous fault-tolerant protocols can be used. Fault-tolerant protocols use the state machine replication algorithm [34] where nodes agree on an ordering of incoming requests. The algorithm has to satisfy two main properties, (1) *safety*: all correct nodes receive the same requests in the same order, and (2) *liveness*: all correct client requests are eventually ordered. Fischer et al. [24] show that in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, in most fault-tolerant protocols, safety is satisfied without any synchrony assumption, however, a synchrony assumption is considered to ensure liveness.

Crash fault-tolerant protocols guarantee safety in an asynchronous network using $2f+1$ nodes to overcome the simultaneous failure of any f nodes while in Byzantine fault-tolerant protocols, $3f+1$ nodes are usually needed to provide the safety property in the presence of f malicious nodes.

We now briefly introduce Paxos [35] and PBFT [19] as two well-known crash and Byzantine fault-tolerant protocols.

In Paxos [35], which guarantees safety in an asynchronous network using $2f+1$ nodes, upon receiving a request from a client, the primary (assuming it is already elected) initiates a consensus protocol among the agents of the applications by multicasting an accept message including the transaction. Once an agent receives an accept message, it sends an accepted message to the primary. The primary waits for f accepted messages from different agents (plus itself becomes $f+1$), multicasts a commit message to all the agents, and sends a reply to the client. Upon receiving a commit message from the primary, each agent executes the transaction.

In the presence of malicious nodes, PBFT [19] can be used. In PBFT, which guarantees safety in an asynchronous network using $3f+1$ nodes, during a normal case execution, a client sends a request to the primary node, and the primary multicasts a pre-prepare message to all agents. Then, during the prepare and commit phases, all agents communicate to each other to reach agreement and send responses back to the client. Note that in Paxos only the primary sends the reply message to the client whereas in PBFT every agent sends the reply message and the client waits for $f+1$ matching reply messages before accepting the result.

Paxos and PBFT can be used to establish consensus among the agents of an application to order internal transactions. To establish consensus for cross-application transactions, since applications may not trust each other, a Byzantine fault-tolerant protocol among applications is needed. To provide both safety and liveness for consensus at the application level, we assume that at most $\lfloor \frac{|A|-1}{3} \rfloor$ applications might be malicious. As a result, to commit a cross-application transaction, by a similar argument as in PBFT [19], at least *two-thirds* ($\lfloor \frac{2|A|}{3} \rfloor + 1$) of the applications *including* the initiator application of each cross-application transaction must agree on the order of the transaction. We need agreement from agents of the initiator application to ensure that the cross-application transaction is consistent with the internal transactions of the initiator application. The conformance between a cross-application transaction and the SLAs is checked by every application during the execution of the transaction, thus, if the initiator application initiates a transaction that does not conform to the SLAs, it will be detected by other applications during the execution.

5. LOCAL CONSENSUS IN CAPER

In this section, we show how internal transactions are ordered and executed in CAPER. CAPER employs *local* consensus within an application to order transactions where the agents of an application, *independent* of other nodes in the network, agree on the order of transactions.

The local consensus protocols in CAPER are pluggable. Depending on the failure model of nodes (agents), the application can use a crash fault-tolerant protocol, e.g., Paxos [35], or a Byzantine fault-tolerant protocol, e.g., PBFT [18], as the local consensus protocol. The application might not even use a consensus protocol and rely on a single non-faulty reliable node to order the transactions. The number of required agents is also determined by the protocol and the maximum number of simultaneous failures in the network.

The local consensus protocol to order the internal transactions of an application is initiated by one of the agents, called *the primary*. The normal case operation for CAPER to execute an internal transaction proceeds as follows. A client c requests an internal transaction tx for an application by sending a message $\langle \text{REQUEST}, tx, \tau_c, c \rangle_{\sigma_c}$ to the agent p of the application it believes to be the primary. Here, τ_c is the client's timestamp and the entire message is signed with signature σ_c. The timestamps of clients are used to totally order the requests of each client and to ensure exactly-once semantics for the execution of client requests.

When the primary p receives a request from a client, it first checks the signature to ensure it is valid, and then initiates a local consensus algorithm by multicasting a message, e.g., accept message in Paxos or pre-prepare message in PBFT, including the requested transaction to other agents. To provide a total order between transactions, the primary also includes $H(t)$ in the message where $H(.)$ denotes the cryptographic hash function and t is the previous transaction that is ordered by the application. If the transaction has a data dependency to a cross-application transaction (as discussed in Section 3.2), the primary includes the cryptographic hash of the cross-application transaction as well.

The agents then establish agreement on a total order of transactions using the utilized consensus protocol, execute the transaction, and append it to the blockchain ledger. Finally, either the primary or every agent node (depending on the local consensus protocol) sends a reply message $\langle \text{REPLY}, \tau_c, u \rangle_{\sigma_o}$ to client c where ts_c is the timestamp of the corresponding request and u is the execution result.

6. GLOBAL CONSENSUS IN CAPER

In this section, we show how cross-application transactions are ordered and executed in CAPER using *global* consensus among applications. We introduce three ordering approaches for achieving global consensus in CAPER.

Ordering transactions using a disjoint set of nodes was introduced by Hyperledger [10] to enhance the scalability of the system and to add flexibility for implementing the consensus protocol, i.e., different protocols can be used to establish consensus. Similarly, in the first approach of CAPER, a disjoint set of nodes, called *orderers*, which are not the agents of any application, are used to globally order cross-application transactions. The global consensus protocol among orderers is pluggable and depending on the specifications of the system, a crash, a Byzantine, or any other fault-tolerant protocol can be used.

In the absence of such orderer nodes, and in the second approach, CAPER relies on the applications to order cross-application transactions in a hierarchical way. To distinguish between trust at the node level and trust at the application level, agreement is established in two levels: a local level among the agents of each application, and a global level among the applications of the system.

Although the second approach does not require a set of orderers to order transactions, the hierarchical nature of the algorithm, which needs local consensus within each application for each step of global ordering, makes the protocol expensive. In the third approach, similar to the second approach, the agents of all applications order cross-application transactions, however, agreement is established in one level.

In all three approaches, when agreement is achieved, the agents execute the transaction and append it to their local views of the ledger. Note that if the transaction does not follow the service level agreements, which are implemented in the public smart contract, it will be detected during the execution of the transaction (as discussed in Section 3.3).

Algorithm 1 Global Consensus using Orderers

1: **init()**:
2: r := *node_id*
3: O := the set of orderer nodes
4: p := the primary agent of the initiator application
5: o := the primary agent of the orderers

6: **upon receiving** m=$\langle$REQUEST, $tx, \tau_c, c\rangle_{\sigma_c}$ and $(r == p)$:
7: if m is *valid* then
8: **initiate** local consensus
9: if *agreement is achieved* then
10: **send** $\langle\langle$ORDER, $h_L, d, r\rangle_{\sigma_r}, m\rangle$ to o ▷ either primary or every agent r of the initiator application

11: **upon receiving** valid $\langle\langle$ORDER, $h_L, d, r\rangle_{\sigma_r}, m\rangle$ from the sufficient number of agents and node is the primary orderer o
12: **initiate** a global consensus among orderers O
13: if *agreement is achieved* then
14: **multicast** $\langle$SYNC, $h_L, h_G, d, o\rangle_{\sigma_o}, m\rangle$ ▷ o or every orderer

15: **upon receiving** $\langle$SYNC, $h_L, h_G, d, o\rangle_{\sigma_o}, m\rangle$ from the primary orderer (or a sufficient number of orderers)
16: **execute and append** the transaction to the ledger

6.1 Global Consensus using a Separate Set of Orderers

Using a separate set of nodes to order transactions adds flexibility to the system by allowing the global consensus implementation to be tailored to the trust assumption of a particular deployment. In addition, since the orderers are decoupled from the agents that execute transactions and maintain the blockchain ledger, using orderers enhances the scalability of the system.

In the first approach, CAPER orders the cross-application transactions using a disjoint set of *orderers* O where for each application α in A, $O \cap N_\alpha = \varnothing$. As discussed earlier, a cross-application transaction is ordered *locally* among the transactions that are initiated by the initiator application and *globally* among all cross-application transactions, thus both local and global orderings are needed. Since orderers are not involved in the local consensus and application agents do not participate in the global consensus, these two orderings are separated from each other. As a result, cross-application transactions are first, similar to the internal transactions, ordered *locally* within each application using the application consensus protocol and then ordered *globally* among all cross-application transactions using orderers. Note that the cross-application transactions are ordered first locally and then globally to prevent the case where the agents of the initiator application do not agree on the local order of a transaction that has already been ordered globally. Once orderers agree on the global order of the transaction, they multicast the transaction to all the applications, thus, every agent of every application executes and appends the transaction to its ledger.

The normal case operation of *global consensus using orderers* to execute a cross-application transaction is presented in Algorithm 1. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-5 of the algorithm, nodes p and o are the primary agent of the initiator application and the primary orderer respectively and O is the set of orderers.

As shown in lines 6-10 of the algorithm, when primary agent p receives a valid request $\langle$REQUEST, $tx, \tau_c, c\rangle_{\sigma_c}$ from an authorized client c (with timestamp τ_c) to execute a cross-application transaction tx, it initiates the local consensus

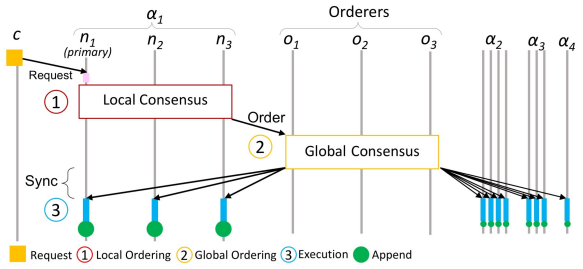


Figure 3: Global Consensus using a Set of Orderers

protocol to establish agreement on the order of the cross-application transaction among all transactions that are initiated by the application. Once agreement is achieved, depending on the local consensus protocol, either the primary or every agent r of the initiator application sends a signed order message $\langle\langle$ORDER, $h_L, d, r\rangle_{\sigma_r}, m\rangle$ including the client's request message m to the primary orderer node where d is m's digest and $h_L = H(t)$ where t is the previous transaction that is ordered by the application. Note that h_L is included in the order messages to let orderers know that the agents of the application agree on the local order of the transaction.

When the order message is sent to the primary orderer, the primary of the application waits for the corresponding reply (sync) message from the orderers before initiating any other transactions. In addition, when local agreement is established, nodes wait for global agreement before appending the transaction to the blockchain ledger.

As shown in lines 11-14, once primary orderer o receives a sufficient number (determined by the local consensus protocol of the initiator application, i.e., 1 for crash fault-tolerant and $f+1$ for Byzantine fault-tolerant) of valid matching order messages (with valid signature, matching h_L, and matching message digest), primary orderer o initiates the (global) consensus protocol by multicasting the transaction to other orderers to establish a total order on cross-application transactions. Once the orderers reach agreement on the order of the transaction, depending on the global consensus protocol, either the primary or every orderer node o multicasts a sync message $\langle$SYNC, $h_L, h_G, d, o\rangle_{\sigma_o}, m\rangle$ to all the agent of every application where d is the digest of m, h_L is copied from order message of the initiator application, and $h_G = H(t)$ such that t is the previous cross-application transaction.

Upon receiving a sync message, the agents of each application log the message. Once an agent receives a *sufficient* number of matching sync messages (determined by the utilized global consensus protocol), the agent executes the transaction and appends the transaction to its blockchain ledger (as can be seen in lines 15 and 16). Note that the agents of the initiator application consider both h_L and h_G hashes to append the transaction to the ledger while the agents of the other applications only consider the hash of the previous cross-application transaction (h_G). Finally, depending on the utilized local consensus protocol, either the primary or every agent node r of the initiator application sends a reply message $\langle$REPLY, $\tau_c, u\rangle_{\sigma_r}$ to client c where ts_c is the timestamp of the corresponding request and u is the result of executing the request.

The flow of cross-application transactions using a set of orderers in a blockchain consisting of four applications α_1, α_2, α_3, and α_4 can be seen in Figure 3. Here application α_1 initiates the cross-application transaction. In addition, o_1,

Algorithm 2 Hierarchical Global Consensus

1: **init()**:
2: $r :=$ *node_id*
3: $\alpha :=$ the application that initiates the consensus
4: $P :=$ the set of primary agents of all applications
5: $p :=$ the primary of α

6: **upon receiving** transaction m and $(r == p)$
7: if m is *valid* then
8: **multicast** $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ to P

9: if $(r == p)$ OR (**upon receiving** $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ from *initiator primary* p and $r \in P$)
10: **initiate** local consensus
11: if *agreement is achieved* then
12: **multicast** $\langle \text{ACCEPT}, h_L, h_G, d, r\rangle_{\sigma_r}$ to P ▷ r or all agents

13: **upon receiving** matching $\langle \text{ACCEPT}, h_L, h_G, d, q\rangle_{\sigma_q}$ from **two-thirds** of the applications including α and $r \in P$
14: **initiate** a local consensus
15: if *agreement is achieved* then
16: **multicast** $\langle \text{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$ ▷ either r or all agents

17: **upon receiving** matching $\langle \text{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$ from **two-thirds** of the applications including α
18: **execute and append** the transaction to the ledger

o_2, and o_3 are the orderer nodes. Upon receiving a cross-application transaction from a client, the primary node n_1 of α_1 validates the transaction and similar to internal transactions, initiates a local consensus algorithm to order the transaction within the application. Once the transaction is internally ordered, an order message is sent to the primary orderer node. The orderers use a global consensus protocol, e.g., a crash fault-tolerant protocol with $f = 1$ in Figure 3, to agree on the global order of the transaction and then since here the orderers are crash-only nodes, the primary orderer (node o_1) multicasts sync messages including the transaction to every agent of every application. Each agent then validates the transaction, executes the transaction, and appends it to the ledger.

Safety and Liveness. Since the global ordering protocol is pluggable, its safety and liveness are implied due to Paxos [35] and PBFT [19]. The order of cross-application transactions on different applications is unique because cross-application transactions are ordered sequentially by orderers and the agents of every application follows the order that is provided by orderers. Note that if the agents of an application do not follow the provided order, the application might not be able to initiate cross-application transactions in the future (since the initiated transactions might not conform to SLAs). To ensure a total order between transactions that are initiated by the same application, once the primary of an application sends a cross-application transaction to be ordered by the orderers, the primary stops initiating any other transactions and waits for the reply from the orderers.

6.2 Hierarchical Global Consensus

While using a separate set of orderer nodes makes the agreement routine simple and modular, it comes with an extra cost of adding orderers to the system. In the absence of such orderer nodes, reaching consensus on the order of the cross-application transactions needs the participation of all applications. To distinguish between trust at the node level and trust at the application level, CAPER uses an asynchronous Byzantine fault-tolerant protocol for the global consensus where for each cross-application transaction and

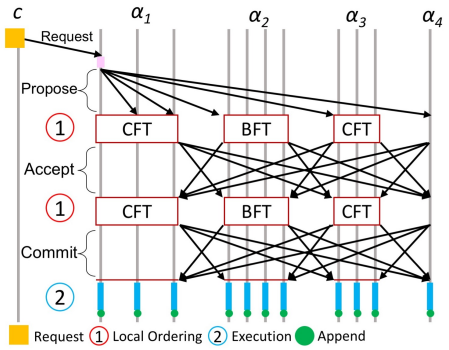


Figure 4: Hierarchical Global Consensus

in each phase of the global consensus, every application runs its local consensus protocol between its agents to internally decide on the application vote in that phase.

In addition, since the agents of the initiator application participate in the global consensus, in contrast to the first approach, the local and global orderings are merged together. However for each step of the global ordering the protocol ensures that the initiator application agrees with the ordering. Hence the transaction is ordered correctly with respect to the transactions that are initiated by the initiator application.

Furthermore, in each step of the global ordering and within each application, once agreement is established, depending on the utilized local consensus protocol, either the primary, e.g., in Paxos, or every agent, e.g., in PBFT, sends the vote to other applications.

Algorithm 2 presents the *hierarchical global consensus*. Same as before, every sent and received message is logged by the agents. As presented in lines 1-5 of the algorithm, P is the set of primary agents of all applications and p is the primary agent of the initiator application α.

Once the primary agent p of the initiator application receives a valid cross-application transaction, as indicated in lines 6-8, it multicasts a signed $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ message to the primary agents of every application where d is the digest of m, $h_L = H(t)$ such that t is the previous transaction that is initiated by the application, and $h_G = H(t')$ such that t' is the previous cross-application transaction. Note that hash h_L is only used by the agents of the initiator application to ensure that the new transaction is ordered correctly with respect to the transactions that are initiated by the application. The agents of the other applications ignore h_L.

As can be seen in lines 9-12, upon receiving a propose message from an application, the primary agent of every application checks the signature, hash h_G, and message digest to ensure the message is valid. Then, every primary agent (including the primary of the initiator application) *internally* initiates the local consensus protocol to establish agreement on the order of the requested transaction. If agreement is achieved, depending on the utilized local consensus protocol, either the primary or every agent multicasts an accept message to the primary agents of all other applications. Note that local consensus is needed to ensures that non-faulty agents agree with the received propose message. Hence, if agreement achieved, they just log the messages and do not append the transaction to their copies of the ledger.

Algorithm 3 One-Level Global Consensus

1: **init()**:
2: r := *node_id*
3: α := the application that initiates the consensus
4: p := the primary agent of α

5: **upon receiving** transaction m and $(r == p)$
6: if m is *valid* then
7: **broadcast** $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ to every agents

8: **upon receiving** $\langle\langle \text{PROPOSE}, h_L, h_G, d\rangle_{\sigma_p}, m\rangle$ from *primary* p
9: if the message is *valid* then
10: **broadcast** $\langle \text{ACCEPT}, h_L, h_G, d, r\rangle_{\sigma_r}$

11: **upon receiving** matching $\langle \text{ACCEPT}, h_L, h_G, d, r\rangle_{\sigma_r}$ from **local-majority** of **two-thirds** of the applications including α
12: if the message is *valid* then
13: **broadcast** $\langle \text{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$

14: **upon receiving** matching $\langle \text{COMMIT}, h_L, h_G, d, r\rangle_{\sigma_r}$ from **local-majority** of **two-thirds** of the applications including α
15: **execute and append** the transaction to the ledger

As shown in lines 13-16, each application waits for valid accept messages from *two-thirds* of the applications *including* the initiator application which are matched with the accept message that is sent by the application. Note that a valid accept message from initiator application is needed to ensure that the transaction is consistent with the transactions that are initiated by that application. Upon receiving a sufficient number of accept messages, the primary agent of each application initiates the local consensus protocol to establish agreement on the received accept messages. Once agreement is achieved, the application (either the primary or every agent) multicasts a commit message to every agent of all other applications.

The propose and accept phases of the global consensus, similar to `pre-prepare` and `prepare` phases of PBFT [19], guarantee that non-malicious applications agree on a total order for the transactions. Indeed, they ensure that no fork happens in the blockchain, i.e., it is not possible to have two different transactions with the same hash h_G. This is true because at least *two-thirds* ($\lfloor \frac{2|A|}{3} \rfloor + 1$) of the applications agreed with the order of each transactions, thus any two quorums of applications intersect in at least $\lfloor \frac{|A|-1}{3} \rfloor + 1$ applications. Since at most $\lfloor \frac{|A|-1}{3} \rfloor$ applications might be malicious, there is at least *one non-malicious* application in the intersection of any two quorums.

Finally, in lines 17 and 18, similar to the previous phase, if an application receives matching commit messages from *two-thirds* of the applications including the initiator one that match the application's commit message, its agents execute the transaction and append it to their ledgers.

Figure 4 shows the hierarchical consensus with four applications (similar to Figure 3) where applications α_1 and α_3 use a crash fault-tolerant (CFT) protocol and application α_2 uses a Byzantine-fault-tolerant (BFT) protocol as their local consensus protocol. Here, the primary of application α_1 initiates the consensus.

As an optimization, for a system with a high percentage of cross-application transactions and to prevent the initiation of concurrent cross-application transactions, the primary node of one of the applications can be designated as a *super primary* where every application sends its cross-application transaction to the super primary and the super primary initiates the protocol.

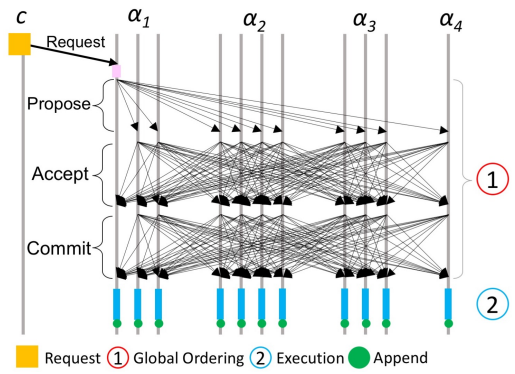


Figure 5: One-level Global Consensus

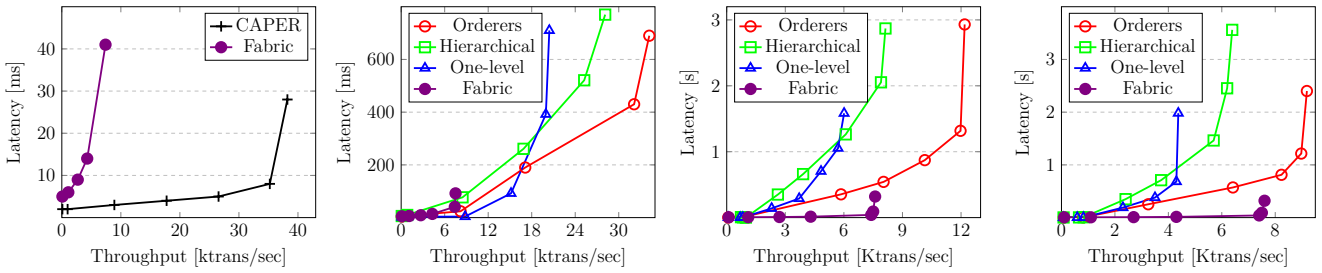
Safety and Liveness. In the hierarchical consensus, as discussed earlier, since at least $\lfloor \frac{2|A|}{3} \rfloor + 1$ of the applications must agree with the order of a transaction and at most $\lfloor \frac{|A|-1}{3} \rfloor$ applications might be malicious, safety is ensured. Indeed, if two or more concurrent transactions are initiated, at most one of them collects the required number of messages (two-thirds of the applications) , i.e., it is not possible for more than one of them to be ordered with the same hash h_G. If none of the concurrent transactions collects enough votes, all initiator applications try to send their transactions again. In such a situation and to ensure liveness, CAPER assigns a timer to each transaction and delays the transactions to prevent concurrent re-initiation of the transactions.

6.3 One-Level Global Consensus

While hierarchical consensus eliminates the need for having an extra set of orderer nodes and also distinguishes between trust at the node level and trust at the application level, it requires an expensive two-level consensus protocol where each step of the global consensus needs the entire local consensus protocol to be run within each application. In this section, we introduce a *one-level* global consensus protocol where for each cross-application transaction, the agents of all applications participate to achieve consensus on the order of the transaction.

Since the number of agents of each application depends on the utilized consensus protocol within the application, the required number of matching replies to ensure that the majority of agents of an application agree on the order of the transaction is different from application to application. Therefore, we define *local-majority* as the required number of matching messages from the agents of an application. If the agents of an application are crash-only nodes, local-majority for the application is equal to $f + 1$ (from the total $2f + 1$ agents), and if the agents of an application might behave maliciously, local-majority for the application is equal to $2f + 1$ (from the total $3f + 1$ agents). For an application that has only a single reliable agent, local-majority is one.

Algorithm 3 presents the *one-level global consensus*. Variable p indicates the primary agent of the initiator application α. As shown in lines 5-7, the primary of the initiator application broadcasts a signed propose message including the transaction, the hash of the previous transaction that is initiated by the application (h_L), and the hash of the previous cross-application transaction (h_G) to the agents of every application. Same as before, h_L is only used by the agents of the initiator application.



(a) 0% Cross-application (b) 20% Cross-application (c) 80% Cross-application (d) 100% cross-application

Figure 6: Throughput/Latency Measurement by Increasing the percentage of Cross-application Transactions

Once an agent receives a propose message, it checks the signature, message digest, and hash h_G to ensure the message is valid. If the agent belongs to the initiator application, it also checks hash h_L. Once the message is validated, the agent broadcasts an accept message to every agent of every application, as indicated in lines 8-10.

As presented in lines 11-13, upon receiving valid accept messages from the local-majority of two-thirds of the applications *including* the initiator application that match the accept message which is sent by the agent, each agent broadcasts a commit message to every agent of every application. The propose and accept phases of the algorithm, similar to `pre-prepare` and `prepare` phases of PBFT [19], guarantee that non-faulty agents agree on a order for the transactions.

Finally, as shown in lines 14 and 15, once an agent receives valid commit messages from local-majority of two-thirds of the applications including the initiator application that matches its commit message, the agent considers the transaction as committed, thus, executes the transaction and appends the transaction to the ledger.

Figure 5 shows the one-level consensus in CAPER for four applications with different failure modes.

Safety and Liveness. To ensure safety in one-level consensus, in each of the accept and commit phases, matching messages from the local majority of two-thirds of the application is required. Local majority of each application is needed to ensure that any two quorums intersect in at least one non-faulty node within the application and two-thirds of the applications is needed to ensure that any two quorums intersect in at least one non-malicious application (since at most $\lfloor \frac{|A|-1}{3} \rfloor$ applications might be malicious). To ensure liveness, similar to the hierarchical consensus, CAPER assigns timers to delay concurrent transactions and also as an optimization uses a super primary for systems with a high percentage of cross-application transactions.

7. EXPERIMENTAL EVALUATIONS

In this section, we conduct several experiments to evaluate CAPER. As explained earlier, CAPER consists of a set of collaborating distributed applications where each application maintains its data in a datastore consisting of private and public records. The private records of the datastore, which are replicated across the agents of application, include the data of internal transactions. The public records, on the other hand, are replicated across every agent of every application and include the data of cross-application transactions. For the purpose of this evaluation, applications are implemented as simple accounting applications where clients can initiate transactions to transfer assets from one or more of their accounts to other accounts.

In addition to CAPER, we also implemented a permissioned blockchain system specifically designed in the execute-order-validate architecture introduced by Fabric [10] where the transactions (internal as well as cross-application) of different applications are executed by the agents (endorsers) of their applications in parallel, ordered by a separate set of orderers, and then validated by every agent of every application. Each block also consists of a single transaction (as in [28]). Note that in the case of Fabric, all internal as well as cross-application transactions are ordered by orderers and all the applications maintain the same blockchain ledger (i.e., the confidentiality of data is not preserved).

The experiments are conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instance with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. In each experiment, we increase the total number of transactions per second from 100 to 100000 (by increasing the number of clients running on a single VM) and measure the end-to-end throughput (x-axis) and latency (y-axis) of the system. The load is equally distributed among the applications.

7.1 Workloads with Cross-Application transactions

In the first set of experiments, we measure the performance of CAPER for workloads with different percentage of cross-application transactions, i.e., 0%, 20%, 80%, and 100%. We consider four applications where each application has three agents and uses a Paxos protocol with $f = 1$ to establish consensus on its internal transactions. To process cross-application transactions we implement all three approaches, using a set of orderers (we refer to this approach as *orderers*), hierarchical, and one-level, which are explained in Section 6. Orderers are implemented using a typical Kafka orderer setup with 3 ZooKeeper nodes, 4 Kafka brokers and 3 orderers (similar to the ordering service of Fabric [10]). The results are shown in Figure 6(a)-(d).

When all transactions are internal (Figure 6(a)), each application processes its transactions independent of other applications. In such a situation, CAPER is able to process upto 36000 transactions (9000 transactions per application) with very low latency (∼10 ms). Here, since there is no dependency between the transactions of different applications and the blockchain views of applications are constructed in parallel, the throughput of the entire system increases lin-

early by increasing the number of applications. With the same latency as CAPER (∼10 ms), Fabric processes 3000 transactions. Fabric is able to process upto 7000 transactions in total with 40 ms latency, however, the end-to-end throughput is saturated beyond 7000 transactions. In Fabric, adding more applications only increases the number of parallel execution threads and since every transaction of all applications is ordered by the same set of orderers, the performance of Fabric is not significantly improved. Note that since all transactions are internal, if Fabric uses different channels for different applications, it can linearly scale as the number of applications increases. However, even with that improvement, CAPER still provides higher throughput (∼29% higher in its peak throughput).

In the second set of experiments, the workload is changed to include 20% cross-application transactions which are equally initiated by different applications. As can be seen in Figure 6(b), when CAPER is not heavily loaded, the one-level consensus approach has better performance since it involves less number of communication phases. As can be seen, the one-level approach processes ∼16000 transactions with 90 ms latency whereas the hierarchical and orderers approaches have latencies of 220 and 150 ms latency respectively in order to process the same number of transactions.

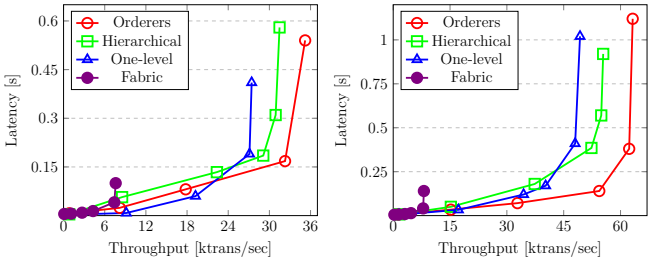
Once CAPER becomes heavily loaded, cross-application transactions might be initiated in parallel. As a result, the latencies of the hierarchical and one-level approaches are dramatically increased (as discussed in Section 6). With 400 ms latency, using orderers, CAPER is able to process 30000 transactions whereas the one-level and hierarchical approaches process 18000 and 19000 transactions (resp.).

The performance of Fabric, however, is not affected by increasing the percentage of cross-application transactions (since all transactions are ordered by the same set of orderers) and Fabric still processes upto 7000 transactions in total with 40 ms latency.

Increasing the percentage of cross-application transactions to 80% (Figure 6(c)) decreases the performance of all three approaches. In this case, using orderers, CAPER is able to process ∼10000 transaction with sub-second latency whereas the hierarchical and one-level approaches process upto 6000 transactions with the same latency. This is expected because in the hierarchical and one-level approaches when the system is heavily loaded, different nodes receive concurrent transactions in different orders.

When all transactions are cross-application, the one-level consensus can only process ∼4000 transactions per second with 800 ms latency whereas using the orderers, CAPER is able to process ∼9000 transactions with the same latency. Fabric, same as before, is able to process upto 7000 transactions in total with 40 ms latency which is better than both hierarchical and one-level approaches.

As mentioned before, since Fabric orders all internal as well as cross-application transactions by the same set of orderers, the performance of Fabric is not affected by increasing the percentage of cross-application transactions. As a result, in workloads with 80% and 100% cross-application transactions, Fabric performs better than all other approaches in terms of latency. In fact, for cross-application transactions and to achieve consensus, CAPER uses either multiple rounds of consensus (in the orderers and hierarchical approaches) or a Byzantine fault-tolerant protocol with a large number of participants (in the one-level approach).



(a) 4 Applications (b) 8 Applications

Figure 7: Performance with 4 and 8 Applications

This is in contrast to Fabric that relies on a single crash fault-tolerant protocol (with only three nodes) to order the transactions which results in a lower latency. However, Fabric does not ensure confidentiality of data in this settings.

Note that even with 100% cross-application transactions, the throughput of CAPER using the orderers approach is slightly higher than Fabric (9% higher in its peak throughput) because the throughput of Fabric is affected by conflicting transactions, i.e., transactions that access the same records, due to its execute-order-validate architecture [9] (in the experiments, ∼10% of the transactions are conflicting).

7.2 Performance with Multiple Applications

In the next set of experiments, we measure the performance of CAPER in two deployments with 4 and 8 applications where each application has three agents and uses a Paxos protocol with $f = 1$ to establish consensus on its internal transactions. For each deployment, we consider workloads with 90% internal and 10% cross-application transactions (the typical settings in partitioned databases [50] [48]).

For the deployment with four applications (Figure 7(a)), the performance of CAPER is close to the scenario in Figure 6(b) where in the workloads with less than ∼20000 transactions per second, the one-level consensus has better performance and beyond that, using orderers is more beneficial. Increasing the number of applications (Figure 7(b)), however, results in higher latency for the same throughput in both one-level and hierarchical consensus due to the increasing number of nodes which increases the chance of conflict between concurrent transactions. Note that since 90% of the transactions are internal, increasing the number of applications improves the overall throughput of CAPER near-linearly (using orderers, CAPER processes upto ∼32000 and ∼59000 transactions per second with four and eight applications (respectively) with 30 ms latency). As mentioned earlier, in Fabric, increasing the number of applications does not significantly improve the performance.

8. RELATED WORK

Most existing permissioned blockchain systems follow the order-execute architecture and differ mainly in their ordering routines. The ordering protocol of Tendermint [32] is different from the original PBFT in two ways. First, only a subset of nodes participate in the consensus protocol and second, the leader is changed after the construction of every block (leader rotation). Quorum [20], as an Ethereum-based [3] permissioned blockchain, introduces a consensus protocol based on Raft [41], a well-known crash fault-tolerant protocol. Quorum, similar to CAPER, supports public and private transactions, however, in Quorum, both public and

private transactions are ordered using the same consensus protocol which results in lower throughput in comparison to CAPER. Quorum also uses cryptography techniques to ensure confidentiality of private transactions. Chain Core [1], Multichain [26], Hyperledger Iroha [5], and Corda [2] are some other prominent permissioned blockchains that utilize order-execute architecture.

Fabric [10], as a permissioned blockchain, introduces the execute-order-validate architecture and leverages parallelism by executing the transactions of different applications simultaneously. Fabric presents modular design, pluggable fault-tolerant protocol, policy-based endorsement, and non-deterministic execution for the first time in the context of permissioned blockchains. In a recent release, Fabric also utilizes the Raft protocol [41] for its ordering service where a leader node is elected (per channel) and replicates messages to the followers. Raft mainly helps organizations to have their own ordering nodes, participating in the ordering service, which leads to a more decentralized system. CAPER utilizes some of the Fabric properties such as modular design and pluggable protocol. In addition, and in contrast to single-channel Fabric, CAPER constructs blocks simultaneously and ensures the confidentiality of both transaction data and ledger, whereas Fabric ensures only transaction data confidentiality using Private Data Collections [7].

Applications deployed on the same or different blockchains need to communicate with each other in order to exchange assets or information. Atomic cross-chain swaps [27] are used for trading assets on two unrelated blockchains. Atomic swaps use hash-lock and time-lock mechanisms to either perform all or none of a cryptographically linked set of transactions. Interledger protocols (ILPV [49]) which are presented by the World Wide Web Consortium (W3C) use a generalization of atomic swaps and enable secure transfers between two blockchain ledgers using escrow transactions. Since the redemption of an escrow transaction needs fulfillment of all the terms of an agreement, the transfer is atomic. Lightning network [39] [43] also generalizes atomic swap to transfer assets between two different clients via a network of micropayment channels. Blocknet [22], BTC [16], Xclaim [53], POA Bridge [6] (designed specifically for Ethereum), Wanchain [8], and Fusion [4] are some other blockchain systems that allow users to transfer assets between two chains.

Fabric also addresses atomic cross-chain swap between permissioned blockchains that are deployed on different channels by either assuming the existence of a trusted channel among the participants or using an atomic commit protocol [11] [10]. CAPER is different from Fabric in two ways: First, cross-chain communications follow the service level agreements and are visible to everyone, and second, CAPER does not need a trusted channel among the participants.

Using sidechain is proposed in [13] to transfer assets from a main blockchain to the sidechain(s) and execute some transactions in the sidechain(s). Sidechains can reduce confirmation time, support more functionality than the main blockchain, and reduce the transaction cost. In sidechains, a set of known nodes, called functionaries, are responsible for moving the assets back from the sidechain to the main chain. Liquid [23], Plasma [42], Sidechains [25], and RSK [37] are some other blockchain systems that use sidechains. Polkadot [52] and Cosmos [33] also construct a main chain which is used by a set of (side) blockchains, i.e., parachains in Polkadot and zones in Cosmos, to exchange value or information. Both Polkadot and Cosmos rely on Byzantine consensus protocols in both sender and receiver sides.

Our work is also related to blockchain systems with Directed acyclic graph structure. Byteball [21] and Iota [44] are two DAG structured permissionless blockchains. In Byteball, a set of privileged users, called witnesses, determines a total order on the DAG to prevent double spending, whereas, in Iota [44], the number of descendant transactions is used to commit a transaction and abort the other one. In Iota, the blockchain, called Tangle, grows in more than one direction. Indeed, once a user issues a transaction, the user must pick two existing transactions and approve them (results in adding edges from the new transaction to the existing ones). The user will also solve a small PoW puzzle similar to Bitcoin. Hashgraph [14] is another DAG structured permissioned blockchain that combines a voting algorithm with a gossip protocol to achieve consensus among nodes. In Hashgraph nodes submit transactions (events) and gossip about transactions by randomly choosing other nodes (neighbors). Each transaction in Hashgraph includes the hash of the previous transactions of both sender and receiver. This process continues until convergence when all nodes become aware of all transactions. Hashgraph, in contrast to CAPER, does not distinguish between internal and cross-application transactions which results in lower performance as well as confidentiality issues. Vegvisir [30], Ghost [46], Inclusive protocol [38], DagCoin [36], Phantom [47], Spectre [45], and MeshCash [15] are some other DAG structured blockchain systems. In CAPER, in contrast to all these blockchains, internal transactions of different applications are added independent of each others and only cross-application transactions need a global consensus. As a result, first, the double spending problem never occurs, and second, internal transactions can be processed simultaneously, which results in lower latency and higher throughput.

9. CONCLUSION

In this paper, we proposed CAPER, a permissioned blockchain system that supports both internal and cross-application transactions of collaborating distributed applications. CAPER targets both performance and confidentiality aspects of blockchain systems. To achieve better performance, CAPER orders and executes internal transactions of different applications simultaneously. In addition, to achieve confidentiality, the blockchain ledger is *not maintained* by any node and each application maintains its own local view of the ledger including its internal and all cross-application transactions. CAPER also distinguishes between trust at the node level and trust at the application level and allows an application to behave maliciously for its benefit while its nodes are non-malicious. Furthermore, CAPER introduces three consensus protocols to globally order cross-application transactions: using a separate set of orderers, hierarchical consensus, and one-level consensus. Our experiments show that for lightly loaded applications one-level consensus shows better performance whereas using a set of orderers is more beneficial for heavily loaded applications. In the absence of extra resources for orderers, the hierarchical approach can provide better performance in heavily loaded applications.

Acknowledgement

This work is funded by NSF grants CNS-1703560 and CNS-1815733.

10. REFERENCES

[1] Chain. http://chain.com.

[2] Corda. https://github.com/corda/corda.

[3] Ethereum blockchain app platform. https://www.ethereum.org. 2017.

[4] Fusion whitepaper: An inclusive cryptofinance platform based on blockchain. https://docs.wixstatic.com/ugd/76b9ac_be5c61ff0e3048b3a21456223d542687.pdf.

[5] Hyperledger iroha. https://github.com/hyperledger/iroha.

[6] Poa bridge. https://github.com/poanetwork/token-bridge.

[7] Private data collections: A high-level overview. https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview.

[8] Wanchain: Building super financial markets for the new digital economy. https://www.wanchain.org/files/Wanchain-Whitepaper-EN-version.pdf.

[9] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019.

[10] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[11] E. Androulaki, C. Cachin, A. De Caro, and E. Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*, pages 111–131. Springer, 2018.

[12] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. Medrec: Using blockchain for medical data access and permission management. In *International Conference on Open and Big Data (OBD)*, pages 25–30. IEEE, 2016.

[13] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille. Enabling blockchain innovations with pegged sidechains. 2014.

[14] L. Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, 1, 2016.

[15] I. Bentov, P. Hubácek, T. Moran, and A. Nadler. Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017:300, 2017.

[16] V. Buterin. Chain interoperability. *R3 Research Paper*, 2016.

[17] C. Cachin and M. Vukolić. Blockchain consensus protocols in the wild. In *31 International Symposium on Distributed Computing, DISC*, pages 1–16, 2017.

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[19] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[20] J. M. Chase. Quorum white paper, 2016.

[21] A. Churyumov. Byteball: A decentralized system for storage and transfer of value. 2016.

[22] A. Culwick and D. Metcalf. The blocknet design specification, 2018.

[23] J. Dilley, A. Poelstra, J. Wilkins, M. Piekarska, B. Gorlick, and M. Friedenbach. Strong federations: An interoperable blockchain solution to centralized third-party risks. *arXiv preprint arXiv:1612.05491*, 2016.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[25] A. Garoffolo and R. Viglione. Sidechains: Decoupled consensus between chains. *arXiv preprint arXiv:1812.05441*, 2018.

[26] G. Greenspan. Multichain private blockchain-white paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[27] M. Herlihy. Atomic cross-chain swaps. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 245–254. ACM, 2018.

[28] Z. István, A. Sorniotti, and M. Vukolić. Streamchain: Do blockchains need blocks? In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 1–6. ACM, 2018.

[29] H. Jin, X. Dai, and J. Xiao. Towards a novel architecture for enabling interoperability amongst multiple blockchains. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1203–1211. IEEE, 2018.

[30] K. Karlsson, W. Jiang, S. Wicker, D. Adams, E. Ma, R. van Renesse, and H. Weatherspoon. Vegvisir: A partition-tolerant blockchain for the internet-of-things. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1150–1158. IEEE, 2018.

[31] K. Korpela, J. Hallikas, and T. Dahlberg. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[32] J. Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 2014.

[33] J. Kwon and E. Buchman. Cosmos: A network of distributed ledgers. *URL https://cosmos. network/whitepaper*, 2016.

[34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[35] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[36] S. D. Lerner. Dagcoin: a cryptocurrency without blocks, 2015.

[37] S. D. Lerner. Rootstock: Bitcoin powered smart contracts, 2015.

[38] Y. Lewenberg, Y. Sompolinsky, and A. Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[39] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and

P. McCorry. Sprites and state channels: Payment networks that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.

[40] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[41] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[42] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.

[43] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.

[44] S. Popov. The tangle. *cit. on*, page 131, 2016.

[45] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

[46] Y. Sompolinsky and A. Zohar. Accelerating bitcoin's transaction processing. *Fast Money Grows on Trees, Not Chains*, 2013.

[47] Y. Sompolinsky and A. Zohar. Phantom: A scalable blockdag protocol, 2018.

[48] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB*, 8(3):245–256, 2014.

[49] S. Thomas and E. Schwartz. A protocol for interledger payments. 2015.

[50] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[51] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using blockchain. In *International Conference on Business Process Management*, pages 329–347. Springer, 2016.

[52] G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 2016.

[53] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. J. Knottenbelt. Xclaim: A framework for blockchain interoperability. 2019.