Rethinking Permissioned Blockchains

[Extended Abstract]

Marko Vukolić
IBM Research - Zurich
mvu@zurich.ibm.com

ABSTRACT

Current blockchain platforms, especially the recent permissioned systems, have architectural limitations: smart contracts run sequentially, all node executes all smart contracts, consensus protocols are hard-coded, the trust model is static and not flexible, and non-determinism in smart-contract execution poses serious problems. Overcoming these limitations is critical for improving both functional properties of blockchains, such as confidentiality and consistency, as well as their non-functional properties, such as performance and scalability.

We discuss these limitations in the context of permissioned blockchains, including an early version of the Hyperledger Fabric blockchain platform, and how a re-design of Hyperledger Fabric's architecture addresses them.

Keywords

Blockchain, Byzantine fault-tolerance, Consensus, Hyperledger Fabric, System architecture

1. BACKGROUND

Permissioned blockchains have evolved as an alternative to permissionless blockchains (in which anybody can participate, e.g., Bitcoin[1], Ethereum[2]), to address the need for running blockchain technology among a set of known and identifiable participants that have to be explicitly admitted to the blockchain network. This concept behind permissioned blockchains is particularly interesting in business applications of blockchain technology and distributed ledgers, in which the participants require some means of identifying each other while not necessarily fully trusting each other.

Permissioned blockchains are quickly gaining a lot of traction across different industries. A good example is the Hyperledger Project[3], a prominent open-source initiative under the auspices of the Linux Foundation, dedicated to bringing blockchain technologies to businesses, with a strong focus on permissioned blockchains. Among over 100 member organizations of the Hyperledger Project, one can find technology companies, fintech startups, world-leading financial organizations (banks, stock-exchanges and financial intermediaries), airplane and consumer electronics manufacturers, telecommunication providers and more.

The state-of-the-art permissioned blockchain systems available today (for example, Kadena[4], Tendermint[5], Chain[6]) typically follow, on a high-level, the design thinking behind permissionless blockchains. However, this leads to suboptimal results more often than not. In particular, a design that works well for public permissionless blockchains built around a cryptocurrency is not necessarily adequate for business applications that want to benefit from distributed ledger technology (DLT) perhaps without dependency on a specific cryptocurrency. We have been confronted with such issues during the development of Hyperledger Fabric, an open-source general-purpose permissioned blockchain system.

Specifically, the limiting design decisions that permissioned blockchains typically inherit from their permissionless relatives include at least: sequential execution of smart contracts performed *after* consensus, execution of smart contracts on *all* nodes, *hard-coded* consensus protocols (whether they use proof-of-work or Byzantine fault-tolerance (BFT) [19]), as well as problems with non-determinism in smart contracts. Apart from these limitations, permissioned blockchains often and unnecessarily expose some low-level trust assumptions originating from their consensus mechanism to their smart-contract applications. This then requires a smart contract to reason about a trust model such as "f faults out of $3f + 1$," which have nothing to do with the application and stem from the low-level BFT consensus protocol.

In this paper, we continue in Section 2 with an overview of these limitations inherent in many permissioned blockchains. Then, in Section 3, we describe how the need for overcoming these limitations has influenced our design decisions for Hyperledger Fabric, and how an architecture re-design in Hyperledger Fabric addresses them. These decisions are aimed at improving both functional properties of Hyperledger Fabric such as confidentiality and consistency, as well as its non-functional properties, such as performance and scalability.

[1]http://bitcoin.org/
[2]http://www.ethereum.org
[3]http://www.hyperledger.org/

[4]http://www.kadena.io
[5]http://tendermint.com
[6]http://chain.com

BCC'17 April 02-02 2017, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-4974-1/17/04.
DOI: http://dx.doi.org/10.1145/3055518.3055526

2. DESIGN LIMITATIONS OF PERMISSIONED BLOCKCHAINS

Sequential execution. Permissionless generic blockchains, such as Ethereum, execute transactions on smart contracts sequentially, usually after consensus or intertwined with it. All permissioned blockchains that we know of inherit this style of execution, which basically follows the active state-machine replication approach, well-known in distributing computing [17, 7]. In this approach, requests to the application (smart contract) are ordered by the consensus and then executed in the same order, sequentially, one request at a time, at all nodes.

This style of execution has several limitations when used in blockchains. Perhaps the biggest one is the bound on the effective throughput that can be achieved by the blockchain. In particular, the throughput becomes inversely proportional to the latency of execution, which may often, except for the simplest smart contracts, become the performance bottleneck. Moreover, an adversary trying to subvert performance of such a blockchain system could simply introduce smart contracts that take a very long time to execute, effectively mounting a denial of service (DoS) attack on the blockchain network.

To cope with this issue, blockchains orchestrated around their own cryptocurrency, such as Ethereum, introduce the concepts such as *gas* which basically requires the transaction (submitter) to pay for every step of the computation performed during smart contract execution. To support the concept, Ethereum goes a long way and introduces its own virtual machine to be able to monitor and control steps of the computation. Whereas this looks like a viable solution for public cryptocurrencies, it is not adequate for many business applications that require the benefits of DLTs, without the actual need for a cryptocurrency. In addition, the need for a specialized VM execution environment limits the languages in which smart contracts can be written, which can also hinder adoption in practice.

Finally, the research literature in the field of distributed systems proposes many directions for overcoming these issues, such as data sharding, parallel execution, or multi-core execution (see, e.g., [19] for specific pointers). Yet many of these need to be re-worked and tailored to the trust model and requirements specific to blockchain.

Non-deterministic execution. One important problem of the active-replication approach are non-deterministic transactions. Indeed, when smart contracts run after consensus on the transaction inputs, their execution must be deterministic; otherwise, the effects of consensus are nullified and execution may result in diverging ledgers or "forks." Ideally, smart-contract execution would always be deterministic; however, this expectation requires smart contract languages and compilers that enforce determinism.

We believe that domain-specific languages (DSLs) for smart contracts are an interesting research topic, ensuring that they are sufficiently expressive for many applications but restricted enough to ensure deterministic execution. However, a more attractive solution would be to program smart contracts in a general-purpose programming language. Such a design would simplify the adoption of blockchain across businesses, as developers would not need to learn specific blockchain DSLs, but could rely on their favorite programming language.

General-purpose languages are problematic with respect to determinism since the application developer does not need to explicitly introduce a clearly non-deterministic operation (such as reading the system time or generating a random number) to produce non-deterministic effects. For instance, in many languages (e.g., golang) a simple map iteration may produce a different order in two executions. For these reasons, tackling non-deterministic execution with trust assumptions and failure models relevant to blockchain is an important research topic [8].

Execution on all nodes. Blockchain smart contracts most often execute on all nodes, following the original public, permissionless blockchains. This is at odds with confidentiality, since for many blockchain use cases, the logic of a smart contract or a transaction input should be restricted to certain nodes.

Whereas cryptography, in particular encryption and zero-knowledge protocols [3], can help to achieve confidentiality, this often comes with a considerable overhead. Without doubt, the zero-knowledge techniques will be useful whenever they do not impair performance, but focusing only on cryptographic techniques for providing confidentiality may be short-sighted.

In principle, the goal of reaching consensus and synchronizing the state across all nodes does not require that all nodes execute all smart contracts. It is sufficient to *propagate the same state* to all nodes. Smart contract execution can thus be restricted to a subset of the nodes trusted for this task, which vouch for the results of the execution, and other nodes simply verify that these results match. Such a design represents a departure from active replication towards a variant of passive replication [6], with one caveat specific to the trust model of blockchain. Namely, which set of nodes can be "trusted" to execute transactions properly, is there a "sufficient" number of them, or should there be more complex way to describe what makes a transaction "valid"? We postpone possible solutions to Section 3, yet this question is tightly related to another limiting factor permissioned blockchains face, which we discuss next.

Trust model flexibility. Permissioned blockchains mostly rely on asynchronous BFT replication protocols to establish consensus [19]. These protocols come with their well-known assumption which stipulates that at least $3f + 1$ nodes are necessary to reach agreement (consensus) in the presence of up to f Byzantine faulty nodes [5]. Moreover, blockchains most often rely on the same nodes for execution that run BFT consensus. Hence, this trust assumption spills over to smart-contract execution as well, even though the required ratio of correct to (potentially) faulty nodes is lower for BFT execution than it is for BFT agreement [20].

Coupled with the executing applications on all nodes, an assumption such as "f out of $3f + 1$" may not match the trust model that a smart-contract developer needs to reason about. Coming back to the question of the "sufficient" number of executing nodes whose execution results must match, one may be tempted to give an "obvious" answer of $f + 1$ (to guarantee execution by at least one correct node). Yet this answer is wrong, not because it is technically wrong, but because it is conceptually wrong, as it exposes a low-level threshold to the application. We firmly believe that permissioned blockchains should decouple application trust assumptions from those pertaining to the underlying consensus protocol and allow smart contract developers to reason

about the execution trust model in a flexible way and independently of the low-level consensus.

Hard-coded consensus. Virtually all blockchain systems of today, whether permissioned or not, come with a hard-coded consensus protocol (a notable exception is Hyperledger Fabric, where consensus has been modular from the start). Changing the consensus protocol in blockchains is very difficult, if not impossible, without serious code rewrites.

This is clearly not optimal, as there is no such "one-size-fits-all" consensus protocol. For instance, BFT protocols are known to exhibit different performances under different system conditions and deployment environments [18]. This is not difficult to see, actually: for instance, a protocol with the "chain" communication pattern that exhibits provably optimal throughput on a LAN cluster with symmetric and homogeneous network bandwidth across different links [10, 11], will typically degrade on a wide-area, heterogeneous network with non-uniform link bandwidth. Furthermore, external conditions such as load, network parameters and current number of faults, may vary over time in a given deployment. This motivates the use of inherently reconfigurable protocol frameworks for BFT consensus, which can adapt to a dynamically changing environment [2]. Another important consideration for matching a conse nsus mechanism to a given blockchain deployment are the trust model and fault assumptions themselves. Indeed, one may want to replace an asynchronous BFT protocol with a protocol in an alternative trust/fault model such as XFT [14], or in some cases even plain crash fault-tolerant (CFT) protocol (e.g., [12, 16]).[7]

With no "one-size-fits-all" consensus protocol, why would one ever build a general-purpose blockchain with a specific hard-coded consensus? Fixing consensus might perhaps make sense if a blockchain is to be deployed only in a static, well-defined environment, in specific use cases that require very fine performance tuning that in turn mandates tight coupling of consensus with other parts of the blockchain system. However, such a design does not scale well across different use cases and deployment scenarios; therefore, a general-purpose permissioned blockchain should be designed with *modular/pluggable* consensus in mind.

3. OVERCOMING THE LIMITATIONS WITH HYPERLEDGER FABRIC

Fabric or, more completely, Hyperledger Fabric (HLF)[8] is an open-source project within the Hyperledger umbrella project. HLF is a modular general-purpose permissioned blockchain system which can be also seen as a distributed operating system for permissioned blockchains.

Starting with its first skeleton version in the beginning of 2016, HLF has supported pluggable consensus. However, it was initially designed in the "classical" way, following the active state-machine replication approach [17] and hence suffering from many of the limitations discussed in Section 2. The active replication architecture remained in place until the v0.6-preview release of Hyperledger Fabric (September 2016). Despite the limitations, v0.6-preview codebase received significant attention by different blockchain stakeholders across industries, as well as in the blockchain research community, with over 100 proof-of-concepts and prototypes built around HLF v0.6-preview, and even with some production deployments (e.g., [15]).

Towards v1 release (due in 2017), and after gaining experiences with earlier versions, the HLF architecture has been overhauled and the system has been re-designed, primarily with the goal of addressing the limitations we outlined in Section 2. Whereas the technical details of the architecture and implementation of Hyperledger Fabric v1 are clearly beyond the scope of this paper, in the rest of this section we briefly discuss how the high-level design of Hyperledger Fabric v1 (hereafter HLFv1) addresses the above limitations [1].

Transition to execute-order-validate architecture. HLFv1 separates nodes responsible for executing *chaincode* (i.e., smart contracts in Hyperledger Fabric parlance) from those responsible for agreement on the order of blocks (consensus). Execution nodes are called *peers*. Each peer maintains a copy of the distributed ledger. Consensus on the order of blocks and transactions in the blockchain is delegated to *orderers* who provide an ordering service without holding distributed ledger state. The semantics of the ordering service are similar to the total-order publish-subscribe service which takes, as an input, transactions from *clients* (producers) and delivers a totally ordered sequence of blocks of transactions to *peers* (consumers).[9]

Before clients submit their transactions to the ordering service, they first submit them for execution to *a subset of* peers that serve as *endorsers* for the transaction and the chaincode. Clients first collect matching signed results of the execution (i.e., versioned updates to the chaincode state) from a "sufficient" number of peers. Then, clients submit the versioned state update, along with gathered signatures, to the ordering service, which in turn outputs a sequence of blocks with such updates to the peers. Finally, all peers perform the *validation* of the endorsement. However, this does not involve re-executing chaincode but merely consists of: (a) verifying the freshness of versioned updates contained in the transaction state update, and (b) verifying signatures from endorsers, to validate that transaction updates were indeed endorsed by "sufficiently" many endorsers.

This approach can be understood as a BFT variant of middleware-based replicated database with asymmetric update processing according to Kemme et al. [13]. It ensures that: (a) chaincode execution *comes before* ordering in HLFv1, and (b) that not all peers execute all chaincodes. This addresses three of the limitations from Section 2:

- Execution needs not necessarily be sequential, as different subsets of peers can execute transactions in parallel. Chaincodes designate only certain peers as endorsers, so different chaincodes can designate different endorsers, allowing for parallel execution.

- Not every peer executes all transactions. While this clearly holds across chaincodes, it sometimes even holds for transactions pertaining to a single chaincode. Namely,

[7]Using CFT may seem at first to be entirely at odds with blockchain trust assumptions; however, CFT consensus is amenable to certain blockchain business use cases in which there is both a need for DLT, as well as a (distributed) trusted (third) party.

[8]https://github.com/hyperledger/fabric

[9]If necessary, peers can also run clients.

a given peer may be required to endorse a given transaction but not another one. In particular, if the transactions are independent (i.e., do not update the same variables) and are allowed to have different endorsers, such transactions may be executed in parallel. Obviously, a chaincode may require that all (or a majority of) the peers executes and endorses its transaction, but this is merely one possible option.

- Disseminating a "sufficient" number of matching state updates helps eliminate the effects of non-determinism [8]. Transaction execution may diverge due to non-determinism, but this is tolerated. If the results of such an execution diverge across replicas, a client will (most probably) not be able to gather the required number of matching replies, and such a non-deterministic transaction might fail. However, non-deterministic execution may never make the state of the peers diverge. This preserves the consistency of HLFv1 in the presence of non-deterministic chaincodes and even allows peers to use local policies to terminate resource-exhaustive execution of DoS transactions.

Flexible endorsement policies. The precise definition of the "sufficient" number of endorsers signatures is stipulated by an *endorsement policy*. The endorsement policy is essentially a transaction validation program that is executed by all peers *after* ordering. However, in HLFv1, endorsement policies are very different from chaincodes, and the application developers cannot code them as they can code chaincodes. A chaincode can simply point to a pre-defined endorsement policy, possibly with some parameters. For instance, a typical endorsement policy will specify that a chaincode has n endorsers (by giving their identities), out of which at least k (chaincode policy parameter) are required to endorse a transaction.

This approach allows for a flexible separation of the trust assumptions for chaincodes from the trust assumptions pertaining to the ordering service (consensus). Chaincode can freely select its endorsement policy (HLFv1 comes with a set of most often used policies) and parameterize it to suit the trust model of the application.

We emphasize that, in principle, an endorsement policy can be an arbitrarily complex validation program, so long as it always produces a deterministic outcome. However, to maintain efficiency and determinism of endorsement policy evaluation (which is done at *all* peers) HLFv1 restricts the number of endorsement policies and their complexities.

Pluggable ordering service (consensus). Finally, as already hinted at, the ordering service is pluggable and modular by design. HLFv1 currently comes with both a CFT and a BFT consensus implementation, as well as with a centralized ordering service (used for development and testing purposes). HLFv1 CFT ordering service is built around Apache Kafka, whereas the BFT ordering service is a modified variant of the PBFT protocol [9], called SimpleBFT. More consensus offering is coming to Hyperledger Fabric; for instance, work is ongoing on integration of the other well-known BFT library to Hyperledger Fabric, namely BFT-SMaRt [4].

ACKNOWLEDGEMENTS

I am very grateful to Christian Cachin, Konstantinos Christidis and Binh Nguyen for their valuable comments and suggestions. This work was supported in part by the EU H2020 project SUPERCLOUD (grant No. 643964) and the Swiss Secretariat for Education, Research and Innovation (contract No. 15.0025).

BIOGRAPHY

Dr. Marko Vukolić is a Research Staff Member at IBM Research - Zurich. Previously, he was a faculty at EURECOM and a visiting faculty at ETH Zurich. He received his PhD in distributed systems from EPFL in 2008 and his dipl. ing. degree in telecommunications from University of Belgrade in 2001. His research interests lie in the broad area of distributed systems, including blockchain and distributed ledgers, cloud computing security, distributed storage and fault-tolerance.

4. REFERENCES

[1] E. Androulaki, C. Cachin, K. Christidis, C. Murthy, B. Nguyen, and M. Vukolić. Hyperledger fabric proposals: Next consensus architecture proposal. http://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md, 2016.

[2] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, Jan. 2015.

[3] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

[4] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 355–362, 2014.

[5] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. 1993.

[7] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[8] C. Cachin, S. Schubert, and M. Vukolić. Non-determinism in byzantine fault-tolerant replication. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, 2016.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[10] R. Guerraoui, N. Knezević, V. Quéma, and M. Vukolić. Stretching bft. EPFL Technical Report 149105, https://infoscience.epfl.ch/record/149105, 2010.

[11] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, 2010.

[12] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, 2011.

[13] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[14] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. XFT: practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 485–500, 2016.

[15] Blockchain News. IBM to launch one of the world's largest blockchain implementations - for IBM global finance. http://www.the-blockchain.com/2016/07/30/ibm-to-launch-one-of-the-worlds-largest-blockchain-implementations-for-ibm-global-finance/, September 2016.

[16] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, 2014.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[18] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 189–204, 2008.

[19] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015*, pages 112–125, 2015.

[20] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 253–267, 2003.