Talking Blockchains: The Perspective of a Database Researcher

Felix Schuhknecht
Institute of Computer Science
Johannes Gutenberg-University Mainz, Germany
schuhknecht@uni-mainz.de

***Abstract*—There are few topics out there, that seem to create as much confusion and discussion as blockchains. This has a multitude of reasons: (1) A large number of drastically different concepts and systems are unified under the very broad term “blockchain”. (2) The topic touches a variety of different fields, including databases, distributed processing, networks, cryptography, and even economics. (3) There exists a large number of different applications of the technology.**

The goal of this paper is to simplify and structure the discussion of blockchain technology. We first introduce a simple formalization of the basic components, that appear again and again in a variety of blockchain systems. Second, we formalize four important execution models, that express the workflow of a large number of blockchain systems. Third, along the way, the we also discuss certain misconceptions, that constantly reappear in discussions. We believe that a common formalization of the transaction processing behavior of blockchain systems both helps beginners to get into the topic as well as can bring experts from different fields to a common denominator when discussing blockchain systems.

I. Introduction

The term “blockchain” describes a highly controversial topic: It unifies a large number of different concepts, systems, and technologies. Many components originate from different fields, such as database systems, distributed processing, computer networks, and cryptography and come together to form a blockchain system. The different perspectives are highly reflected in the literature about the topic, as each field has a different view on the realized concepts: While papers in the database community typically focus on the transaction processing workflow of the system [6], [7], [9], [11], [16], [17], a paper published in the distributed systems community might focus on the consensus mechanism [8], [18], [19] instead. Further, across papers, different terminology is used for the same concepts: For instance the blockchain is called transaction log, history, ledger, or ordered sequence of blocks, depending on who references it. Additionally, different underlying properties and requirements are often assumed. Of course, each field also has a different view on the individual importances of the components. All this renders the discussion, comparison, and generalization of blockchain technology unnecessarily hard and leads to a fragmentation of the blockchain research community.

During our recent work on the topic [16], [17], which is deeply rooted in the database community, we felt a strong need for a more unified discussion of the topic. In the following paper, we thus present a formalization of basic conceptual components, that appear repeatedly in blockchain systems (Section II). Further, based on these components, we present a simple formalization of four execution models, that are implemented by a variety of blockchain systems (Section III). The goal of these models is to enable the discussion of the transaction processing behavior of blockchain systems on the conceptual level, without interweaving the discussion with implementation details and architectural designs. Along the way, we additionally discuss certain misconceptions, that constantly reappear and unnecessarily confuse the audience.

II. Components and their Relation

A. The Network and its Organizations.

We begin on the top-most architectural level with the *network*. A network $N = \{Org_0, ..., Org_{n-1}\}$ is a set of n *organizations*, which are able to communicate with each other. We have to differentiate between *trusted* and *untrusted* networks. A trusted network is a network, where all organizations are assumed to behave *honestly*. This is the type of network, that is typically assumed by classical distributed database systems. In contrast to that, in an untrusted network, not all organizations are assumed to behave honestly – some organizations might behave *arbitrarily*. This is the type of network, that is faced and tolerated by blockchain systems. In the following, when using the term “network”, we mean “untrusted network”.

As we have seen, the type of network is related to the behavior of its organizations. An organization behaves honestly, if it follows the *intended behavior*, i.e. it follows the specification of the system. Likewise, an organizations behaves arbitrarily, if it somehow deviates from the intended behavior.

Misconception 1: Unfortunately, an organization is often treated as a concrete physical instance, such as a node, a peer, or a machine. Instead, we advocate that an organization should be seen as a logical instance, which has the sole purpose of defining the border of trust: Between organizations, there is no trust – within an organization, there is trust. Apart of that, the formalization does not restrict how an organization is physically realized: Within one organization, a single machine could hold all data. Within another organization, the data could be partitioned across a distributed DBMS of 100 nodes. It is simply not important for the behavior of the blockchain system. This abstraction is nicely realized by the blockchain framework Tendermint [3], which completely decouples the behavior of the network from the concrete implementation within an organization, which is up to the user of Tendermint. For example, the project BigchainDB [2] combines Tendermint with the distributed document store MongoDB [4] to implement an organization.

B. The Ledger and Immutability.

Independent from the concrete realization of an organization, conceptually, each organization Org_i has to maintain a copy of the ledger L_i (aka the blockchain) as its primary and most important data structure.

Misconception 2: This full replication of the ledger across all organizations is often listed as a strong feature of blockchain systems – everyone owns all data. However, it is actually a necessary evil that stems from the ideal of full decentralization: If an organization wants to be able to decide on its own, whether a transaction is valid or not, then it needs a local copy of the ledger to do so.

At the core, the ledger $L = [B_0, ..., B_{j-1}]$ is nothing but an an ordered sequence of j *blocks* containing transactions. The very first block B_0 in the ledger is typically called the genesis block and bootstraps the initial state.

Misconception 3: Often, it is suggested that grouping transactions in blocks is an essential part of the system – however, forming blocks is merely a performance optimization. Exchanging larger chunks of memory between organizations results in overall less communication and cryptographic overhead than processing at the granularity of transactions. Fixing the block size to one transaction essentially eliminates the concept of blocks entirely.

The ledger only supports a single modification, namely the appending of the next block, via $L.append(B_j) = L'$. Immutability mechanisms ensure that the deletion, modification, or insertion of a block in the middle of the ledger requires changing all subsequent blocks as well. Precisely, a block $B_i = (H_i, [T_0, ..., T_{k-1}])$ consists of a *header* H_i and an ordered sequence of k *transactions*. To enforce immutability, each block header maintains a fingerprint (aka cryptographic hash) of the previous block, i.e. $H_i = f_B(B_{i-1})$. Often, the header also stores a fingerprint of all transactions within the block. In this setup, it is sufficient to keep the fingerprint of the previous block header, i.e. $H_i = (f_B(H_{i-1}), f_T([T_0, ..., T_{k-1}]))$, which allows a verification of the ledger integrity, by inspecting only the headers (as e.g. used by Bitcoin's Simplified Payment Verification [14]).

Misconception 4: Immutability of the ledger and the mechanisms to enforce it are often listed as a core feature of blockchain systems – what makes it into the ledger cannot be altered retrospectively (or only with very high effort). Again, we should see immutability more as a counter-measure, that is required in the presence of arbitrarily behaving organizations, and less as a feature: In the case of Bitcoin, the mechanism highly decreases the chances of success for double-spending attacks [14]. Besides, one could realize the used immutability mechanisms not only in blockchain systems, but also to secure the transaction log of standard DBMSs.

A transaction $T = [o_0, ..., o_{l-1}]$ is an ordered sequence of l operations. An operation simply describes either a reading access to the value of a key k as r_k or a writing access of the pair (k, v) as $w_{k,v}$.

Misconception 5: Smart contracts are essentially stored procedures [10], a well-known feature in DBMSs since ages. However, smart contracts can be embedded in the ledger, rendering them immutable after being deployed. They represent a sequence of deterministic [1], [3] or non-deterministic [6], [16] operations $[o_0, ..., o_{l-1}]$, that is installed once and which can be invoked (and parametrized) by transactions. This indirection is part of the implementation of the system and irrelevant on the conceptual level, as transactions invoking smart contracts are part of the ledger. Thus, we simply omit smart contracts in our formalization.

C. The State and Execution.

Additionally to the ledger L_i, we assume that each Org_i contains a state S_i. Thus, $Org_i = (L_i, S_i)$. Note that the state is an auxiliary data structure, that can be recomputed from the ledger at all times.

For the highest flexibility, we use a key-value model in the formalization as the underlying representation. This grants us highest flexibility. Of course, we can easily express structured data, such as in the relational model, within the key-value model. In total, the state $S = \{(k_0, v_0, ver_0), ..., (k_{m-1}, v_{m-1}, ver_{m-1})\}$ is a set of m key-value pairs along with a version number, where obviously $k_0 \neq ... \neq k_{m-1}$. An operation can be executed on the state, which potentially generates a new state, if the operation is a write. Thus, $S.exec(w_{k,v}) = S'$ respectively $S.exec(r_k) = S$. Consequently, we also support the execution of a transaction via $S.exec(T) = S.exec(o_0).exec(o_1)... .exec(o_{l-1})$ and the execution of a block via $S.exec(B) = S.exec(T_0).exec(T_1)... .exec(T_{k-1})$

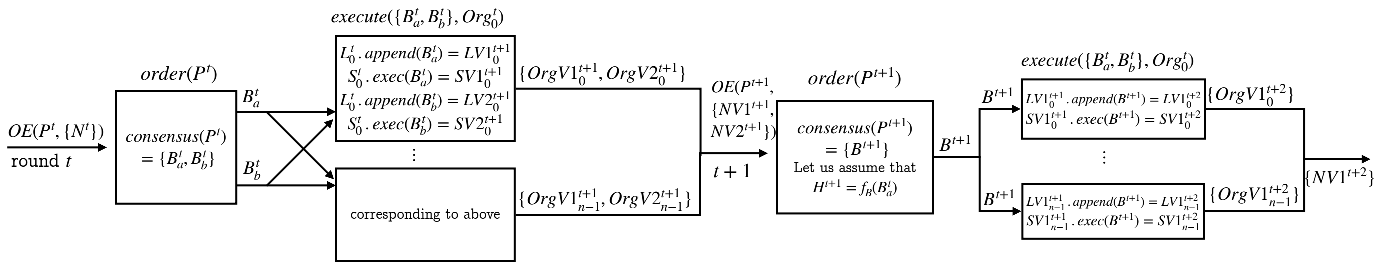
D. Clients and the Pool.

Apart from the network and its organization, there are two other concepts involved in basically any system: Clients and the transaction pool. A client C can propose transactions to the network. Typically, but not necessarily, a client is affiliated with an organization. All proposed transactions are kept in the pool $P = \{T_0, ..., T_{p-1}\}$, where they are pending for processing. Just like an organization, we treat the pool as a purely logical concept: It is globally present, and we leave open how it is physically realized.

III. Execution Models

The previously defined components are the basic building blocks of our formalization. In the following, let us see how they interact in certain execution models.

Misconception 6: Heavily interweaving the transaction processing workflow and the behavior of the consensus mechanisms [13] unnecessarily complicates implementations and



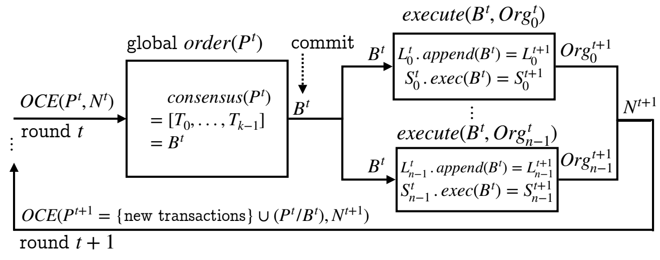
$execute(\{B_a^t, B_b^t\}, Org_0^t)$
$order(P^t)$
$OE(P^t, \{N^t\})$
round t
$consensus(P^t) = \{B_a^t, B_b^t\}$
B_a^t
B_b^t
$L_0^t.append(B_a^t) = LV1_0^{t+1}$
$S_0^t.exec(B_a^t) = SV1_0^{t+1}$
$L_0^t.append(B_b^t) = LV2_0^{t+1}$
$S_0^t.exec(B_b^t) = SV2_0^{t+1}$
corresponding to above
$\{OrgV1_0^{t+1}, OrgV2_0^{t+1}\}$
$\{OrgV1_{n-1}^{t+1}, OrgV2_{n-1}^{t+1}\}$
$OE(P^{t+1}, \{NV1^{t+1}, NV2^{t+1}\})$
$t+1$
$order(P^{t+1})$
$consensus(P^{t+1}) = \{B^{t+1}\}$
Let us assume that $H^{t+1} = f_B(B_a^t)$
B^{t+1}
$execute(\{B_a^t, B_b^t\}, Org_0^t)$
$LV1_0^{t+1}.append(B^{t+1}) = LV1_0^{t+2}$
$SV1_0^{t+1}.exec(B^{t+1}) = SV1_0^{t+2}$
$LV1_{n-1}^{t+1}.append(B^{t+1}) = LV1_{n-1}^{t+2}$
$SV1_{n-1}^{t+1}.exec(B^{t+1}) = SV1_{n-1}^{t+2}$
$\{OrgV1_0^{t+2}\}$
$\{OrgV1_{n-1}^{t+2}\}$
$\{NV1^{t+2}\}$

Fig. 2. The working principle of the Order-Execute model (OE) for the case where two blocks B_a^t and B_b^t are created in the same round t. The block B^{t+1} of the next round builds upon B_a^t and thus resolves the fork of the previous round.

renders the system less flexible. Fabric [6], which implements the EOVC model (Section III-C) took the right measure and was the first system to support a pluggable consensus mechanism. We believe transaction processing and consensus are orthogonal and as such, we will separate them in the formalization of all models. Note that our execution models operate in *rounds*. To denote the status of an individual component X in round t, we write X^t in the following. E.g. P^2 describes the content of the pool in round 2.

A. Order-Commit-Execute (OCE)

The most straight-forward replicated processing model is the Order-Commit-Execute model (OCE). Figure 1 visualizes the formalization of the OCE model. As the name suggests,



global $order(P^t)$
commit
$execute(B^t, Org_0^t)$
$OCE(P^t, N^t)$
round t
$consensus(P^t) = [T_0, \ldots, T_{k-1}] = B^t$
B^t
$L_0^t.append(B^t) = L_0^{t+1}$
$S_0^t.exec(B^t) = S_0^{t+1}$
Org_0^{t+1}
$execute(B^t, Org_{n-1}^t)$
$L_{n-1}^t.append(B^t) = L_{n-1}^{t+1}$
$S_{n-1}^t.exec(B^t) = S_{n-1}^{t+1}$
Org_{n-1}^{t+1}
N^{t+1}
$OCE(P^{t+1} = \{\text{new transactions}\} \cup (P^t/B^t), N^{t+1})$
round $t+1$

Fig. 1. The working principle of the Order-Commit-Execute model (OCE).

it operates in three phases: (1) In the global order-phase, the organizations of the network try to agree on processing a specific block of transactions, that are pending in the pool P^t. To do so, some kind of consensus mechanism is used. (2) As soon as they agree on a specific block B^t, the block is logically committed. (3) The block B^t is distributed to every organization in the network. Each organization Org_i validates B^t, appends it to its ledger L_i^t and executes the transaction of the block against its state S_i^t. This produces a new ledger L_i^{t+1} and a new state S_i^{t+1} . Formally, round t of the OCD model can be described as

$$OCE(N^t = \{Org_0^t, ..., Org_{n-1}^t\}, P^t) = (N^{t+1}, P^{t+1})$$

with $order(P^t) = consensus(P^t) = B^t$ and $N^{t+1} = \{execute(B^t, Org_0^t), ..., execute(Org_{n-1}^t)\}$ and $P^{t+1} = \{\text{new transactions}\} \cup (P^t/B^t)$. The pool P^{t+1} of the next round consists of the newly arrived transactions as well as the transactions, that are left over from the previous round. Note that the OCE model does not produce forks, a linear evolving of the ledger is guaranteed. A system, which implements the OCE model, is for example Tendermint [3] or version 0.6 of Hyperledger Fabric.

B. Order-Execute (OE)

A relaxation of the OCE model is the Order-Execute model (OE). Figure 2 visualizes the working principle. It differs from the OCE-model in that it omits an explicit committing of blocks before the execution across organizations. Instead, the order-phase is allowed to produce *multiple* candidate blocks for the current round t (two blocks in our visualization). This creates a fork of the ledger and a rule must be specified, that determines which branch of the ledger defines the truth. E.g. the rule could be that the current longest branch defines the truth. The order-phase of the subsequents round(s) then determines which of the candidate blocks of the current round eventually becomes a block of the truth branch by building upon it. As a consequence, a block needs to reach a certain depth in the ledger to be considered as committed.

This OE model is implemented by a large number of public systems, including Bitcoin [14] and Ethereum [1].

C. Execute-Order-Validate-Commit (EOVC)

A problem of the previously described OCE and OE models is that they do not scale with the number of organizations, as each organizations has to execute each and every transaction. The Execute-Order-Validate-Commit model (EOVC) tackles this problem by allowing a speculative parallel execution of transactions. Conceptually, it works in four phases: (1) In the simulate-phase, each organization Org_i receives a transaction from the pool and simulates its operations against its state S_i^t. This produces a set capturing all performed reads and writes, without modifying the state. (2) The network agrees on a specific order of the previously executed transactions. (3) As the execution happened speculatively *before* the ordering, we might have conflicts between transactions: E.g. if a transaction T_{42}, that is ordered before a transaction T_{43}, updates a key-value pair to a new version, that was read by T_{43} in the previous version, then T_{43} was executed on a outdated value and must not commit. Thus, the purpose of the validate-phase is to identify all conflicts within the block and mark all valid transactions. (4) In the final commit phase, each organization

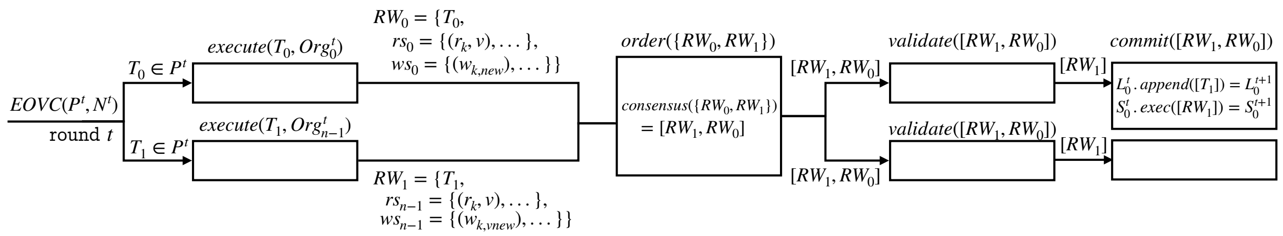

$RW_0 = \{T_0,$
$rs_0 = \{(r_k, v), \dots\},$
$ws_0 = \{(w_{k,new}), \dots\}\}$
$execute(T_0, Org_0^t)$
$T_0 \in P^t$
$EOVC(P^t, N^t)$
round t
$execute(T_1, Org_{n-1}^t)$
$T_1 \in P^t$
$RW_1 = \{T_1,$
$rs_{n-1} = \{(r_k, v), \dots\},$
$ws_{n-1} = \{(w_{k,vnew}), \dots\}\}$
$order(\{RW_0, RW_1\})$
$consensus(\{RW_0, RW_1\})$
$= [RW_1, RW_0]$
$[RW_1, RW_0]$
$validate([RW_1, RW_0])$
$[RW_1]$
$commit([RW_1, RW_0])$
$L_0^t.append([T_1]) = L_0^{t+1}$
$S_0^t.exec([RW_1]) = S_0^{t+1}$

Fig. 3. The working principle of the Execute-Order-Validate-Commit (EOVC) model.

serially updates its original state S_i^t by the write set of all valid transactions.

The EOVC model is implemented by Hyperledger Fabric [6]. It implements the model with an additional support for policies, that specify which organization have to simulate a particular transaction. In particular, workloads with high contention impose a problem for this execution model and several works address this challenge [5], [12], [15], [17].

D. Order-Execute-Commit aka Whatever-Voting (WV)

In the OCE model (Section III-A), consensus was reached on which block to execute. This assumes that the execution happening afterwards happens in the same way across all organizations. The Whatever-Voting (WV) model [16] does not make this assumption: The idea is to push the consensus towards the end of the round. This allows the model to relax the assumptions on the behavior of anything happening *before* consensus. Figure 4 visualizes the concept.

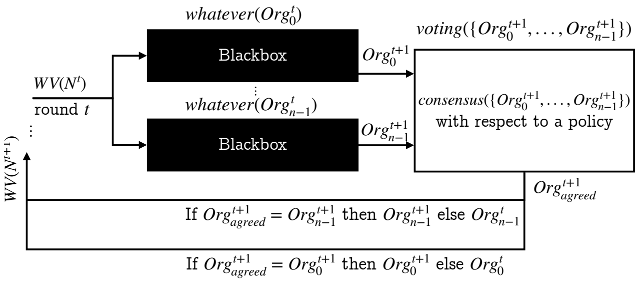

$whatever(Org_0^t)$
Blackbox
$whatever(Org_{n-1}^t)$
Blackbox
$WV(N^t)$
round t
$WV(N^{t+1})$
$voting(\{Org_0^{t+1}, \dots, Org_{n-1}^{t+1}\})$
Org_0^{t+1}
Org_{n-1}^{t+1}
$consensus(\{Org_0^{t+1}, \dots, Org_{n-1}^{t+1}\})$
with respect to a policy
Org_{agreed}^{t+1}
If $Org_{agreed}^{t+1} = Org_{n-1}^{t+1}$ then Org_{n-1}^{t+1} else Org_{n-1}^t
If $Org_{agreed}^{t+1} = Org_0^{t+1}$ then Org_0^{t+1} else Org_0^t

Fig. 4. The working principle of the Order-Execute-Commit model.

The model works in two phases: (1) In the W-phase, each organization does whatever it deems necessary to pass the consecutive voting phase. Typically, but not necessarily, this involves participating in a global ordering process, executing the transactions of the block against the state, and updating the ledger. In (2) the V-phase, it is checked whether a consensus can be reached on the produced outcomes with respect to a policy. If it can be reached, each organization that in line with the consensus logically commits ledger and state and proceeds.

WV is implemented by our project chainifyDB [16], that allows form a network of different blackbox DBMSs.

IV. Conclusion

In this paper, we have presented a simple formalization of the basic building blocks and four executions models of blockchain systems. We focused on the conceptual transaction processing behavior to avoid interweaving the logical workflow with implementation details and architectural design choices. Further, we identified certain misconceptions, that appear again and again in blockchain discussions. We believe that, although our description resembles the perspective of a database researcher, it can serve as the basis of a more unified discussion on the topic in the future.

References

[1] Ethereum: https://github.com/ethereum/wiki/wiki/white-paper.
[2] Bigchaindb: https://www.bigchaindb.com, 2019.
[3] Tendermint: https://tendermint.com/, 2019.
[4] Mongodb: https://www.mongodb.com, 2020.
[5] M. J. Amiri, D. Agrawal, and A. E. Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 1337–1347. IEEE, 2019.
[6] E. Androulaki, A. Barger, V. Bortnikov, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018.
[7] C. Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.
[8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186. USENIX Association, 1999.
[9] H. Dang, T. T. A. Dinh, D. Loghin, E. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *SIGMOD Conference*, pages 123–140. ACM, 2019.
[10] A. Eisenberg. New standard for stored procedures in SQL. *SIGMOD Rec.*, 25(4):81–88, 1996.
[11] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb - a shared database on blockchains. *PVLDB*, 12(11):1597–1608, 2019.
[12] C. Gorenflo, L. Golab, and S. Keshav. XOX fabric: A hybrid approach to blockchain transaction execution. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*, pages 1–9. IEEE, 2020.
[13] S. Gupta, J. Hellings, and M. Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain.* Morgan & Claypool Publishers, 2021.
[14] S. Nakamoto. Bitcoin: https://bitcoin.org/bitcoin.pdf.
[15] P. Ruan, D. Loghin, Q. Ta, M. Zhang, G. Chen, and B. C. Ooi. A transactional perspective on execute-order-validate blockchains. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 543–557. ACM, 2020.
[16] F. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal. chainifydb: How to get rid of your blockchain and use your dbms instead. *CIDR*, 2021.
[17] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Conference*, pages 105–122. ACM, 2019.
[18] M. Zbierski. Parallel byzantine fault tolerance. In *ACS*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 321–333. Springer, 2014.
[19] W. Zhao. Optimistic byzantine fault tolerance. *IJPEDS*, 31(3):254–267, 2016.