Consensus on Demand

Jakub Sliwinski, Yann Vonlanthen, and Roger Wattenhofer

ETH Zürich

Abstract. Digital money can be implemented efficiently by avoiding consensus. However, no-consensus designs are fundamentally limited, as they cannot support general smart contracts, and similarly they cannot deal with conflicting transactions.
We present a novel protocol that combines the benefits of an asynchronous, broadcast-based digital currency, with the capacity to perform consensus. This is achieved by selectively performing consensus a posteriori, i.e., only when absolutely necessary. Our on-demand consensus comes at the price of restricting the Byzantine participants to be less than a one-fifth minority in the system, which is the optimal threshold. We formally prove the correctness of our system and present an open-source implementation, which inherits many features from the Ethereum ecosystem.

Keywords: Consensus · Reliable broadcast · Blockchain · Fault tolerance · Cryptocurrency.

1 Introduction

Following the famed white paper of Satoshi Nakamoto [30], a plethora of digital payment systems (cryptocurrencies) emerged. The basic functionality of such payment systems are money transfer transactions. These transactions are stored in a distributed ledger, a fault-tolerant and cryptographically secured append-only database. Most cryptocurrencies have a ledger where transactions are *totally ordered*, effectively forcing all participants of the system to perform the state transitions sequentially. This sequential verification of all transactions is considered the main bottleneck of distributed ledger solutions [12].

However, in reality, most transactions have no dependencies between each other. For example, a transaction from Alice to Bob and a transaction from Charlie to Dani can be performed in any order. Verifying such independent transactions in parallel offers a vast efficiency improvement. Indeed, recent research proposes *"no-consensus"* payment systems that do not order independent transactions [29, 12]. Such systems can achieve unbounded transaction throughput, as all transactions can be verified in any order, in parallel.

However, no-consensus payment systems suffer from fundamental limitations, as they lack the means to deal with conflicting inputs: If Charlie sets up two

transactions, one for Alice, one for Bob, but Charlie does not have enough funds for both transactions, no-consensus payment system might end up in a deadlock, with Charlie ultimately losing access to her account, and neither Alice nor Bob getting paid. The same problem fundamentally prevents no-consensus systems from supporting general smart contracts, where many uncoordinated parties might issue conflicting inputs to the same smart contract at the same time.

We are faced with a choice: either we use a total ordering currency which cannot scale to a high transaction throughput, or we use a parallel no-consensus verification system that is functionally restricted, and cannot resolve conflicting transactions.

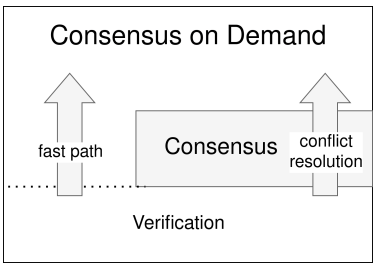

Fig. 1. A high level overview of our protocol.

In this work we propose a system which combines the advantages of both approaches. Our system offers all of the benefits of no-consensus systems, such as in principle unbounded throughput and powerful resiliency to network attacks. Our design first tries to verify every transaction without performing consensus. Only if a transaction cannot be verified on this "fast path", we invoke a consensus routine to resolve potential conflicts.

Our contributions are as follows:

1. We present a protocol we call ConsensusOnDemand. Assuming access to an existing consensus protocol, ConsensusOnDemand is a wrapper algorithm where the first phase offers the benefits of no-consensus systems. In situations where conflicting inputs cannot be processed by pure no-consensus systems (and only those situations), ConsensusOnDemand invokes the consensus instance to resolve the deadlock. The wrapper protocol is resilient to completely asynchronous network conditions as long as $n > 5f$, where n is the total number of participants and f is the number of Byzantine participants. The common case (no consensus) is optimal with regard to latency and does not rely on complex broadcast primitives.
   Thus, we combine the power of processing unrelated inputs in parallel with the ability to resolve conflicting inputs when needed and pave the way for implementations of systems with unbounded throughput and full smart contract functionality.
2. We exhibit our idea in the context of online payments. We describe our protocol, including the pseudocode, and prove the algorithm's correctness.
3. We implement our design as a digital currency following a no-consensus approach enhanced with consensus on demand. A smart contract is used as the example consensus instance. Our implementation is built on top of the

Ethereum client *go-ethereum*, and thus features a network discovery protocol and advanced wallets, while being compatible with the Ethereum ecosystem.

2 Model

We distinguish between clients and servers. Clients are free to enter and leave the system as they please. Servers are in charge of securing the system. We assume that the set of servers Π is fixed and known to all servers.

Clients and servers that follow the protocol are said to be honest. Byzantine clients or servers are subject to arbitrary behavior and might collude when attempting to compromise the system's security. We assume there are no more than f Byzantine servers and that the set of Byzantine servers is static. Further, let $n = |\Pi|$. We assume that $n > 5f$, in other words, less than one-fifth of servers are Byzantine.

Servers are connected all-to-all with authenticated links. Communication is **asynchronous** i.e. messages are delivered with arbitrary delays. We assume standard cryptographic primitives to hold, more specifically, MACs and signatures cannot be forged.

Finally, the model might have to be restricted further in order to reflect the assumptions needed for the choice of the underlying consensus instance. For the consensus algorithm chosen for our implementation (see Section 8) we indeed assume synchronous communication.

3 Problem Statement

We formulate the problem in the context of a cryptocurrency. Initially, the state of the system consists of a known assignment of currency amounts to clients. The system's purpose is to accept transactions, where a transaction $t = (sender, sn, recipient, amount)$ moves an *amount* of currency from a *sender* to a *recipient*. Each client can issue transactions as the sender, where the sequence number sn starts from 0 and increases by 1 for each transaction.

Definition 1. *Two transactions t and t' are said to **conflict**, if they have the same sender and sequence number but $t \neq t'$, i.e., the recipient or the amount differ.*

Existing broadcast-based payment systems [12, 3] provide the guarantees of a Byzantine reliable broadcast for every transaction:

Definition 2. *Each honest server observes transactions from a set of conflicting transactions $\{t_0, t_1, ...\}$. **Byzantine reliable broadcast** satisfies the following properties:*

1. ***Totality****: If some honest server accepts a transaction, every honest server will eventually accept the same transaction.*
2. ***Agreement****: No two honest servers accept conflicting transactions.*
3. ***Validity****: If every honest server observes the same transaction (there are no conflicting transactions), this transaction will be accepted by all honest servers.*

The totality and agreement properties guarantee consistent state of the system and that at most one transaction per unique $(sender, sn)$ pair can be accepted, thus preventing double-spends.

Validity ensures that if the client issued only one transaction for a given sequence number, the transaction will indeed be accepted. However, otherwise the definition does not guarantee termination. In other words, if the client issues conflicting transactions, the system might deadlock and never decide on accepting any of them.

Through the use of this weak abstraction, broadcast-based payment systems combine many benefits, such as resilience to complete asynchrony and fast acceptance. The standout advantage is perhaps the inherent ability to parallelize the processing of independent transactions, resulting in unbounded throughput through horizontal scaling [5, 29].

The crucial assumption that well-behaved clients will not issue conflicting transactions is warranted for a rudimentary payment system. However, it inherently precludes more advanced applications where conflicting inputs naturally occur, such as uncoordinated parties issuing conflicting inputs to a smart contract. To support the full range of blockchain applications, a stronger guarantee needs to hold:

Definition 3. *Each honest server observes transactions from a set of conflicting transactions* $\{t_0, t_1, ...\}$. ***Consensus*** *satisfies the following properties:*

1. *All properties of Byzantine reliable broadcast, and*
2. ***Termination:*** *Every honest server eventually accepts one of the observed transactions.*

The objective of this work is to combine the benefits of broadcast-based designs with the power of consensus: a) non-conflicting transactions are to be processed in a broadcast-based fashion: each honest server broadcasting one acknowledgement for a transaction is enough to accept it; and b) consensus is supported to resolve conflicts.

4 Related Work

Broadcast-based Protocols In 2016 Gupta [20] points out that a payment system does not require consensus. Later, Guerraoui et al. [19] prove that the consensus number of a cryptocurrency is indeed 1 in Herlihy's hierarchy [21].

Table 1. A comparison of existing solutions and ConsensusOnDemand (CoD). The CoD wrapping of a consensus is asynchronous and leaderless, and thus any potentially stronger assumptions are inherited from the consensus instance being used.

	Bitcoin and Ethereum [30]	*Ouroboros [22]*	*Algorand [17]*	*PBFT [10]*	*Red Belly [13]*	*BEAT [15]*	*Broadcast-based [12]*	*CoD + PBFT*	*CoD + BEAT*
Energy-efficient		✓	✓	✓	✓	✓	✓	✓	✓
Deterministic finality			✓	✓	✓	✓	✓	✓	✓
Permissionless	✓	✓	✓						
Leaderless					✓	✓	✓		✓
Asynchronous						✓	✓		✓
Parallelizable							✓	✓	✓
Consensus	✓	✓	✓	✓	✓	✓		✓	✓

Both Guerraoui et al. [12] and Baudet et al. [5] propose a payment scheme where the ordering of transactions is purely determined by the transaction issuer. In their simplest form those currencies rely on Byzantine reliable broadcast, as originally defined by Bracha and Toueg [8]. Srikanth and Toueg [38] as well as Bracha [7] propose well-known Byzantine reliable broadcast algorithms with $\mathcal{O}(n^2)$ message complexity per instance. We use Bracha's Double-Echo algorithm [7] as a fundamental building block and comparison to our approach.

The Cascade protocol [36] promises similar benefits, while being permissionless, i.e., participants are free to enter and leave the system as they please.

Other approaches have proposed a probabilistic Byzantine reliable broadcast [18]. By dropping determinism, efficiency is gained, more specifically $\mathcal{O}(n \log(n))$ messages are shown to be sufficient for each transaction. Our implementation relies on a practical and widely adopted probabilistic broadcast protocol.

Instead, it is possible to drop the *totality* property of Byzantine reliable broadcast and build a payment system where servers distribute themselves proof (a list of signatures) that they are indeed in the possession of the claimed funds. This was also proposed by Guerraoui et al. [12], based on a digital signature approach inspired by Malkhi and Reiter [27]. The message complexity is hereby improved to $\mathcal{O}(n)$.

Remedying the Consensus Bottleneck Early work by Pedone et al. [32] and Lamport [25] recognizes that *commuting* transactions do not need to be ordered

in the traditional state machine replication (SMR) problem with crash failures. Follow-up protocols also deal with Byzantine faults and show fundamental lower-bounds [26, 33, 35].

Removing global coordination in favor of weaker consistency properties also receives a lot of attention outside the area of state machine replication. Conflict-free Replicated Data Types (CRDT) [9, 34] provide a principled approach to performing concurrent operations optimistically, and have recently also been applied to permissioned blockchains [31].

It is often tricky to compare protocols, as they can differentiate themselves in one of the many dimensions, such as synchrony, fault-tolerance and fast path latency [6]. A recent protocol called Byblos [6] achieves 5-step latency in a partially synchronous network when $n > 4f$. Suri-Payer et al. [39] improve the fast path latency to 2 communication steps, in the absence of Byzantine behavior.

Kursawe's optimistic Byzantine agreement protocol [23] features a fast path paired with a consensus protocol in the slow path, with each component being modular. While Kursawe's proposed fast path requires synchronous rounds and no Byzantine failures to happen, our protocol features the same optimal fast path of a single round-trip, while not relying on synchrony and tolerating f Byzantine servers. This comes at the cost of requiring $n \geq 5f + 1$ servers. This bound has been shown to be optimal by Martin et al. [28]. Kuznetsov et al. [24] have recently shown the lower bound to be $n \geq 5f - 1$ in the special case where the set of *proposers* (clients) is a subset of *acceptors* (servers). Their insight is to disregard the acknowledgement of a provably misbehaving server. Although we do not assume the required special case, as in our model the set of clients is external to servers and changing freely, the assumption might well be warranted in other contexts, wherein their approach is applicable to our work.

Our protocol improves upon the solutions of Kursawe and Kuznetsov et al. by being leaderless and asynchronous even in the slow path. This is crucial as leader-based protocols have been shown to be susceptible to throughput degradation in the case of even one slow replica [1, 2, 11, 15, 42]. Song et al. [37] solution is probably most similar to ours, as their Bosco algorithm provides the same decision latency as ours. However, their solution does not focus on reducing the number of invocations of the underlying consensus, meaning that consensus is still performed for every decision.

Sharding is the process of splitting a blockchain architecture into multiple chains, allowing parallelization as each chain solves the state replication task separately. The improvement brought forward in this area [4] is orthogonal to the one we address in this work. Indeed, while having multiple shards allows systems to parallelize operations overall, inside each shard transactions still need to be processed sequentially.

Implementations Recent systems that remove or reduce the need for consensus have shown great promise in terms of practical scalability. More specifically,

Astro [12] is able to perform 20,000 transactions per second, in a network of 200 nodes, with transactions having a latency of less than a second. A similar system by Spiegelman et al. [14] that uses consensus without creating overhead achieves 160,000 tx/sec with about 3 seconds latency in a WAN. The Accept system [29] scales linearly, and has been shown to achieve 1.5 Million tx/sec.

5 A Simple Payment System

We describe a digital currency called BroadcastCoin that serves as a foundation. The protocol disseminates transactions through separate instances of Byzantine reliable broadcast. Crucially, the protocol does not rely on transactions being executed sequentially.

As explained in Section 3, clients start with a given account balance. Clients can access a server to submit transactions $t = (sender, sn, recipient, amount)$. We assume that all transactions are signed using public-key cryptography and that servers only handle transactions with valid signatures. Clients can go offline whenever they please, but are required to keep track of the number of transactions they have performed so far, in order to choose correct, i.e. increasing, sequence numbers.

The BroadcastCoin algorithm determines the agreed order of transactions of a given client to be executed. A transaction accepted by the underlying Byzantine reliable broadcast instance is executed (i.e., the funds are moved) as soon as all previous transactions belonging to the corresponding sender are executed, and enough funds are available in the sender's balance.

The **BroadcastCoin** interface of a server (bc) exports the following events:

- **Request**: $\langle bc.\text{Transfer} \mid s, sn, r, a\rangle$: Allows a client s to submit a transaction with sequence number sn sending a units of cryptocurrency to a recipient client r.
- **Request**: $\langle bc.\text{RequestBalance} \mid c\rangle$: Retrieves the amount of cryptocurrency client c currently owns.
- **Indication**: $\langle bc.\text{Balance} \mid c, a\rangle$: Amount a of cryptocurrency currently owned by client c.

In Byzantine reliable broadcast algorithms, a transaction t typically undergoes the following steps before being accepted:

1. *Dissemination*: A server broadcasts t received by a client by sending it to all servers.
2. *Verification*: Servers acknowledge t if they have never acknowledged a conflicting transaction t'.
3. *Approval*: Servers that receive more than $\frac{n+f}{2}$ acknowledgements for a transaction, broadcast an `APPROVE` message. Servers also broadcast an `APPROVE` message, if they see more than $f + 1$ approvals. A server that receives more than $2f + 1$ approvals, accepts the transaction.

Algorithm 1 BroadcastCoin

```
1: Uses:
2:     Authenticated Perfect Point-to-Point Links, instance al
3:     Byzantine Reliable Broadcast, instance rb
4:
5: upon event ⟨bc.Init | initialDistribution⟩ do
6:     currentSN := [](−1);                          ▷ dictionary initialized with -1
7:     pending := {};                                                  ▷ empty set
8:     balance := initialDistribution;                                 ▷ dictionary
9:
10: upon event ⟨bc.RequestBalance | client⟩ do
11:     trigger ⟨bc.Balance | client, balance[client]⟩;
12:
13: upon event ⟨bc.Transfer | [sender, sn, recipient, amount]⟩ do
14:     t := [sender, sn, recipient, amount];
15:     trigger ⟨rb.Broadcast | [sender, sn], t⟩;          ▷ will be changed in Section 6
16:
17: upon event ⟨rb.Deliver | [sender, sn], t⟩ do
18:     pending[t.sender] = pending[t.sender] ∪ t;
19:
20: upon ∃t ∈ pending such that isValidToExecute(t) do
21:     balance[t.sender] = balance[t.sender] − t.amount;
22:     balance[t.recipient] = balance[t.recipient] + t.amount;
23:     currentSN[t.sender] = currentSN[t.sender] + 1;
24:     pending[t.sender] = pending[t.sender] \ t;
25:
26: procedure isValidToExecute(t) is
27:     return currentSN[t.sender] = t.sn − 1 ∧ balance[t.sender] ≥ t.amount;
28:
```

6 Consensus on Demand

This section presents the core of our contribution that improves upon BroadcastCoin by providing higher functionality as well as lower latency in the fast path. The Byzantine reliable broadcast instance *rb* is substituted by two steps. A best-effort broadcast primitive is used to disseminate transactions efficiently. Then the first transaction t for a given $(sender, sn)$ received by a server is the input value proposed in the corresponding ConsensusOnDemand instance. ConsensusOnDemand uses an underlying consensus instance to provide conflict resolution when necessary. We stress that the combination of the broadcast and consensus steps can be implemented in a variety of ways. The version we present in the following consists of best-effort broadcast paired with consensus as defined in Definition 3, while in Section 7 we mention a different combination. As before, a transaction traverses three stages:

1. *Dissemination:* The transaction is broadcast to all servers.

2. *Verification:* Servers issue an acknowledgement for the first valid transaction they observe for a given $(sender, sn)$ pair. If at any point, a server observes a quorum of more than $\frac{n+3f}{2}$ acknowledgements for a transaction t, the server accepts t.
3. *Consensus (opt.):* If after receiving $n - f$ acknowledgements servers observe conflicting acknowledgments, they propose the transaction for which they have observed the most acknowledgements up to this point to the consensus instance identified by the $(sender, sn)$ pair. The transaction decided by the consensus routine is then accepted immediately, if the transaction hasn't already been accepted by the fast path.

Note that the first stage is identical to the first stage in the Byzantine reliable broadcast considered in Section 5. Although the acceptance condition is also similar, it is performed without the additional broadcast round of `APPROVE` messages. This means that in the common case, transactions are accepted with less delay. The final stage consists of performing consensus if necessary.

The crux of this construction is that a transaction accepted by the fast path should never conflict with a transaction accepted in the slow path. This holds true, since if a transaction t can be accepted by an honest server in the fast path, even though conflicting transactions exists, then every other honest server is guaranteed to observe a majority of acknowledgements for t in a quorum of size $n - f$. Thus, all honest servers will propose t to the underlying consensus instance, and by its validity property, every server will eventually also accept t.

Fig. 2 illustrates this argument in the case where $n = 5f + 1$. There are $3f + 1$ honest servers that acknowledge t and f honest servers that acknowledge t'. By issuing acknowledgements for t, the adversary could bring some servers to accept the transaction t in the fast path. Hence, ConsensusOnDemand should never accept t'. This can be guaranteed, as every quorum containing more than $n-f$ servers (such as Q_1) has a majority of servers acknowledging t. Thus, every server will propose t to the consensus instance, which will accept t due to its validity property. Theorem 3 proves this intuition.

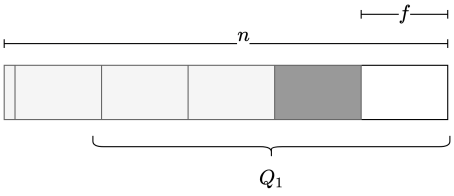

Fig. 2. The two shades of gray represent the share of honest servers acknowledging t (light gray) and t' (dark gray). The adversary is depicted in white, and can acknowledge either transaction. While a server might see more than $4f$ acknowledgments for t, no server sees a majority of acknowledgements for t' in a quorum of $n - f$ servers.

Algorithm 2 ConsensusOnDemand

```
1: Implements:
2:     Consensus, instance fc for the (sender, sn) tuple
3:
4: Uses:
5:     Consensus, instance con
6:     Authenticated Perfect Point-to-Point Links, instance al
7:
8: upon event ⟨fc.Init⟩ do
9:     accepted, con_proposed := False;
10:    acks := [n](⊥);                          ▷ array of size n initialized with ⊥
11:
12: upon event ⟨fc.Propose | t⟩ do
13:    for all q in Π do
14:        trigger ⟨al.Send | q, [ACK, t]⟩
15:    end for
16:
17: upon event ⟨al.Deliver | p, [ACK, t]⟩ do
18:    if acks[p] = ⊥ then
19:        acks[p] := t;
20:    end if
21:
22: upon exists t ≠ ⊥ such that #({p ∈ Π | acks[p] = t}) ≥ (n+3f)/2 and accepted =
    False do
23:    accepted := True;
24:    trigger ⟨fc.Accept | t⟩;
25:
26: upon exists p, q ∈ Π such that acks[p] ≠ acks[q] and #({p ∈ Π | acks[p] ≠
    ⊥}) ≥ n − f and con_proposed = False do
27:    majority := argmax_{t∈T}(#({p ∈ Π | acks[p] = t})
28:    con_proposed := True;
29:    trigger ⟨con.Propose | majority⟩
30:
31: upon event ⟨con.Accept | t⟩ such that accepted = False do
32:    accepted := True;
33:    trigger ⟨fc.Accept | t⟩;
34:
```

Theorem 1. *ConsensusOnDemand satisfies Validity.*

Proof. If every honest server observes the same transaction t, then every honest server broadcasts an acknowledgment for t. Thus every server is guaranteed to eventually observe at least $n-f$ acknowledgements for t. Since $f < \frac{n}{5}$, it follows that $\frac{n+3f}{2} < n - f$, thus every server eventually accepts t.

Theorem 2. *ConsensusOnDemand satisfies Termination.*

Proof. If every honest server observes the same transaction t, by the same argument as in Theorem 1, every server accepts t in a single message round-trip. If an honest server observes too many conflicting acknowledgments to accept a transaction on the fast path, then at least two honest servers have issued conflicting transactions. Hence, eventually, every correct server will propose a transaction to the consensus instance *con*. By *termination* of consensus, *con* will eventually accept a transaction, and thus every honest server will eventually accept a transaction.

Theorem 3. *ConsensusOnDemand satisfies Agreement.*

Proof. First, let us assume that a server accepts a transaction t without using the consensus instance. This means that the server has seen more than $\frac{n+3f}{2}$ acknowledgments for t. This implies that more than $\frac{n+3f}{2} - f = \frac{n+f}{2}$ honest servers have acknowledged t.

Before proposing, every server waits for the arrival of $n-f$ acknowledgements, out of which at least $n-2f$ come from honest servers. Together, both sets contain more than $\frac{n+f}{2}+n-2f = \frac{3(n-f)}{2}$ acknowledgements coming from honest servers. However, there are no more than $n-f$ honest servers, meaning that both sets have more than $\frac{3(n-f)}{2} - (n-f) = \frac{n-f}{2}$ acknowledgements in common. This implies that acknowledgements for t will be the most received acknowledgement at every honest server.

Therefore, every honest server will either accept t though its fast path or, if there is a conflicting transaction, propose t to the consensus instance. Due to its validity property, no honest server will accept a value different from t, thus satisfying agreement.

If no server observes more than $\frac{n+3f}{2}$ acknowledgments for a single transaction, then all honest servers will fall back to the consensus instance, and due to its agreement property, the agreement of ConsensusOnDemand is also satisfied.

7 Discussion

Throughput. No-consensus payment systems have been shown to scale linearly with more computing resources [12, 5]. In particular the simple design of Mathys et al. [29] can be directly applied as the implementation of the fast path of our design, and their result supports our claim that the fast path of our protocol has in principle unbounded throughput.

Slow path abuse. In ConsensusOnDemand, malicious clients can increase the likelihood that consensus needs to be performed by submitting conflicting transactions intentionally.

Due to the completely asynchronous communication model, in our protocol servers keep listening for potentially conflicting acknowledgements of past

transactions that might trigger a consensus invocation. This requirement can be avoided by replacing best-effort broadcast in the fast path with (probabilistic) reliable broadcast. In this configuration, servers for which the fast path succeeds do not have to participate in the slow path at all, as thanks to reliable broadcast's totality property, every honest server is guaranteed to eventually be able to complete the fast path. This modification would make it harder for malicious clients to intentionally invoke consensus, but on the other hand a more complicated broadcast primitive would be used (two echo rounds instead of one).

Intentional abuse of the slow path can also be addressed through game-theoretic means. Economic incentives, such as fees, can be set up so that intentional consensus invocation is adequately costly for a malicious client. The subject of incentive schemes in blockchain systems is broad, as different aspects of the system's functionality need to be considered depending on the application. It is thus left outside the scope of this paper.

Fast path-only synchronization. We presented ConsensusOnDemand in the form where the consensus outcome is accepted by the servers without further steps. Consider the following addition to our protocol. Suppose a server s has not acknowledged a transaction t in the fast path, and later t is the result of consensus. Even though s might have acknowledged a conflicting transaction t' in the fast path, let now s broadcast a fast path acknowledgement for t. By introducing this rule, we ensure that all honest servers that observe the consensus outcome additionally issue a fast path acknowledgement. Afterwards, all accepted transactions can be determined only following the fast path condition.

In this setup, any records of consensus performed by the system can be forgotten, as any agent synchronizing with the state of the system conveniently only needs to be supplied with the fast path acknowledgements.

8 Implementation

We implement the BroadcastCoin protocol described in Algorithm 1 by utilizing the core of the *go-ethereum* client for Ethereum. The main modules that are of relevance are briefly described below.

Transactions: There are two types of transactions in Ethereum. We only support transactions that lead to message calls, and do not support transactions that lead to contract creation. Transactions are broadcast using the Ethereum Wire Protocol [16] that probabilistically disseminates blocks through gossip with a sample size of $\sqrt{n}$.

Blocks: The fundamental building blocks of Ethereum also lay at the core of our protocol. However, instead of using a single chain of blocks to totally order transactions, blocks are used to broadcast batches of acknowledgments. This is done by including all transactions that should be signed in a block created by the server. The block signature proves the authenticity of all acknowledgments.

The *parentHash* field of a block is also kept, in order to refer to the previous block, which allows for easier *synchronization* between servers.

Blockchain: As every server issues its own chain of blocks, we re-purpose the blockchain abstraction to keep track of all chains in a DAG and allow for synchronization with new clients in future extensions.

Mining: We replace the proof-of-work engine with our own logic that determines which transactions from the transaction pool are safe to be acknowledged. Acknowledgements are batched in blocks, signed and broadcast every 5 seconds.

Transaction pool: The transaction pool module is managing new transactions in Ethereum. Most functions and data structures shown in the pseudocode of Algorithm 1 are closely matching the implementation of this module.

We complete the implementation of our protocol by enhancing the no-consensus payment system with the ConsensusOnDemand algorithm. We do so by plugging in a simple consensus algorithm built on the Ethereum Rinkeby testnet. More specifically, we provide a smart contract that is able to perform consensus for any $(sender, sn)$ instance. The contract terminates either when $f + 1$ equal proposals for t are collected, in which case it immediately accepts t. Alternatively, once $2f + 1$ proposals are collected, the contract accepts the most frequent input. The smart contract is called *Multishot* and its implementation can be found in [41], while Appendix A shows the pseudocode and the correctness proof of this algorithm.

While our algorithm is agnostic to the underlying consensus algorithm used, this simple smart contract allows us to demonstrate the effectiveness of ConsensusOnDemand, while keeping our implementation in the Ethereum ecosystem.

These few modules make up most of the changes that were required for us to leverage a large part of the existing *go-ethereum* infrastructure. This allows us to take advantage of the network discovery protocol [16] and the support for hardware wallets. Moreover, our server can simultaneously function as a client, which can be controlled through a variety of interfaces. While the regular JavaScript console can be used, the client can also be addressed via a standard web3 JSON-RPC API accessible through HTTP, WebSockets and Unix Domain Sockets. The complete infrastructure is open source [40].

References

1. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Prime: Byzantine replication under attack. Dependable and Secure Computing, IEEE Transactions on **8**, 564 – 577 (09 2011)
2. Antoniadis, K., Desjardins, A., Gramoli, V., Guerraoui, R., Zablotchi, I.: Leaderless consensus. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). pp. 392–402 (2021)
3. Auvolat, A., Frey, D., Raynal, M., Taïani, F.: Money transfer made simple: a specification, a generic algorithm, and its proof. arXiv preprint arXiv:2006.12276 (2020)

4. Avarikioti, G., Kokoris-Kogias, E., Wattenhofer, R.: Divide and scale: Formalization of distributed ledger sharding protocols. arXiv preprint arXiv:1910.10434 (2019)
5. Baudet, M., Danezis, G., Sonnino, A.: Fastpay: High-performance byzantine fault tolerant settlement. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 163–177 (2020)
6. Bazzi, R., Herlihy, M.: Clairvoyant state machine replication. Information and Computation p. 104701 (2021)
7. Bracha, G.: Asynchronous byzantine agreement protocols. Information and Computation **75**(2), 130 – 143 (1987)
8. Bracha, G., Toueg, S.: Asynchronous Consensus and Broadcast Protocols. JACM **32**(4) (1985)
9. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: Specification, verification, optimality. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 271–284. POPL '14, Association for Computing Machinery, New York, NY, USA (2014)
10. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) **20**(4), 398–461 (2002)
11. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. p. 153–168. NSDI'09, USENIX Association, USA (2009)
12. Collins, D., Guerraoui, R., Komatovic, J., Kuznetsov, P., Monti, M., Pavlovic, M., Pignolet, Y.A., Seredinschi, D.A., Tonkikh, A., Xygkis, A.: Online payments by merely broadcasting messages. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 26–38. IEEE (2020)
13. Crain, T., Natoli, C., Gramoli, V.: Red belly: a secure, fair and scalable open blockchain. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 466–483. IEEE (2021)
14. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the 17th European Conference on Computer Systems. pp. 34–50 (2022)
15. Duan, S., Reiter, M.K., Zhang, H.: Beat: Asynchronous bft made practical. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 2028–2041. CCS '18 (2018)
16. Foundation, E.: Ethereum wire protocol (eth). https://github.com/ethereum/devp2p/blob/master/caps/eth.md (2021)
17. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th symposium on operating systems principles. pp. 51–68 (2017)
18. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., Seredinschi, D.A., Vonlanthen, Y.: Scalable byzantine reliable broadcast (extended version). arXiv preprint arXiv:1908.01738 (2019)
19. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M., Seredinschi, D.A.: The consensus number of a cryptocurrency. Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing - PODC '19 (2019)
20. Gupta, S.: A non-consensus based decentralized financial transaction processing model with support for efficient auditing. Arizona State University (2016)
21. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) **13**(1), 124–149 (1991)

22. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Annual international cryptology conference. pp. 357–388. Springer (2017)
23. Kursawe, K.: Optimistic byzantine agreement. In: 21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings. pp. 262–267. IEEE (2002)
24. Kuznetsov, P., Tonkikh, A., Zhang, Y.X.: Revisiting optimal resilience of fast byzantine consensus. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. pp. 343–353 (2021)
25. Lamport, L.: Generalized consensus and paxos (2005)
26. Lamport, L.: Lower bounds for asynchronous consensus. Distributed Computing **19**(2), 104–125 (2006)
27. Malkhi, D., Reiter, M.: A high-throughput secure reliable multicast protocol. Journal of Computer Security **5**(2), 113–127 (1997)
28. Martin, J.P., Alvisi, L.: Fast byzantine consensus. IEEE Transactions on Dependable and Secure Computing **3**(3), 202–215 (2006)
29. Mathys, M., Schmid, R., Sliwinski, J., Wattenhofer, R.: A Limitlessly Scalable Transaction System. In: 6th International Workshop on Cryptocurrencies and Blockchain Technology (CBT), Copenhagen, Denmark (September 2022)
30. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf (2009)
31. Nasirifard, P., Mayer, R., Jacobsen, H.A.: Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains. In: Proceedings of the 20th International Middleware Conference. p. 110–122. Middleware '19 (2019)
32. Pedone, F., Schiper, A.: Generic broadcast. In: International symposium on distributed computing. pp. 94–106. Springer (1999)
33. Pires, M., Ravi, S., Rodrigues, R.: Generalized paxos made byzantine (and less complex). Algorithms **11**(9), 141 (2018)
34. Preguiça, N.: Conflict-free replicated data types: An overview. arXiv preprint arXiv:1806.10254 (2018)
35. Raykov, P., Schiper, N., Pedone, F.: Byzantine fault-tolerance with commutative commands. In: International Conference On Principles Of Distributed Systems. pp. 329–342. Springer (2011)
36. Sliwinski, J., Wattenhofer, R.: Asynchronous proof-of-stake. In: 23rd International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS) (November 2021)
37. Song, Y.J., Renesse, R.v.: Bosco: One-step byzantine asynchronous consensus. In: International Symposium on Distributed Computing. pp. 438–450. Springer (2008)
38. Srikanth, T.K., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing **2**(2), 80–94 (Jun 1987)
39. Suri-Payer, F., Burke, M., Wang, Z., Zhang, Y., Alvisi, L., Crooks, N.: Basil: Breaking up bft with acid (transactions). In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 1–17 (2021)
40. Vonlanthen, Y.: Cascadeth. https://github.com/yannvon/cascadeth (2021)
41. Vonlanthen, Y.: Multishot. https://github.com/yannvon/aposteriori (2021)
42. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356 (2019)

Appendix A Implementation Details

We visualize the core pipeline of our implementation in Figure 3. The functions and data structures shown closely match our changes to the *go-ethereum* transaction pool module, where most of the logic regarding the acceptance of a transaction is situated.

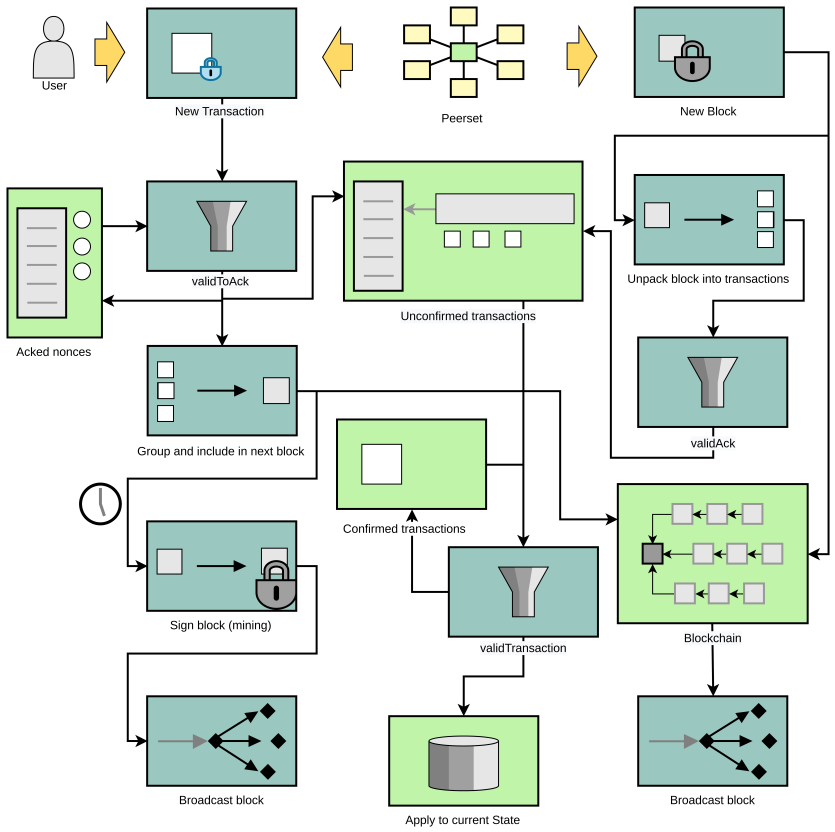

Fig. 3. The core pipeline of our server implementation. The lighter rectangles represent data structures, while the dark rectangles are network or computation operations.

As an underlying source of consensus an Ethereum smart contract is used. Written in Solidity, its pseudocode for a single instance is described by Algorithm 3.

We show that the contract satisfies consensus under the assumption that the Ethereum blockchain itself does not revert and indeed itself provides consensus.

Agreement is trivially satisfied through the use of a designated leader (smart contract).

Termination is satisfied since if all $n \geq 4f+1$ honest servers propose, there are guaranteed to be $2f+1$ proposals, upon which the algorithm terminates.

Validity is satisfied, since if all honest servers propose the same transaction t, then every sample of $f+1$ proposals contains at least one proposal for t. Further, no matter what the majority proposal is, either it was proposed more than f times, in which case validity is satisfied through the previous argument, or, more than f proposals were not t, in which case two honest servers have proposed different values. Hence, we are guaranteed to satisfy validity by accepting the majority value.

Algorithm 3 MultishotConsensus

Implements:
    Consensus, **instance** con for the (sender, sn) tuple

upon event $\langle con.\text{Init} \rangle$ **do**
    $proposals := [n](\bot)$; ▷ Array of size n.
    $accepted :=$ `False`;

upon event $\langle con.\text{Propose} \mid u, [sender, sn], t \rangle$ **do**
    $proposals$[u] := t;

upon exists $t, sender, sn \neq \bot$ **such that** $\#(\{u \in \Pi \mid proposals[u] = t\}) \geq f+1$ **and** $accepted =$ `False` **do**
    $accepted :=$ `True`;
    trigger $\langle con.\text{Accept} \mid [sender, sn], t \rangle$

upon exists $sender, sn \neq \bot$ **such that** $\#(\{u \in \Pi \mid proposals[u] \neq \bot) \geq 2f+1$ **and** $accepted =$ `False` **do**
    $accepted :=$ `True`;
    t := $argmax_{t \in T}(\#(\{u \in \Pi \mid proposals[u] = t\})$
    trigger $\langle con.\text{Accept} \mid [sender, sn], t \rangle$