# Wendy, the Good Little Fairness Widget - Achieving Order Fairness for Blockchains

Klaus Kursawe

klaus@vega.xyz

## ABSTRACT

The advent of decentralized trading markets introduces a number of new challenges for consensus protocols. In addition to the 'usual' attacks – a subset of the validators trying to prevent agreement – there is now the possibility of financial fraud, which can abuse properties not normally considered critical in consensus protocols. We investigate the issues of attackers manipulating or exploiting the order in which transactions are scheduled in the blockchain. More concretely, we look into *order fairness*, i.e., ways we can assure that the order of transactions relative to each other is fair. We show that one of the more intuitive definitions of fairness is impossible to achieve. We then present Wendy, a group of low overhead protocols that can implement different concepts of fairness. Wendy acts as an additional widget for an existing blockchain, and is largely agnostic to the underlying blockchain and its security assumptions, as long as they provide a known and always active set of validators. Furthermore, it is possible to implement fairness for some subsets of the transactions, and thus run several independent fair markets (as well as some unfair ones) on the same chain.

## 1 INTRODUCTION

In the last years, blockchain applications have increased in complexity and utility, allowing more advanced financial tools such as exchanges and trading markets to be decentralized. The introduction decentralized trading markets introduces a number of new challenges for consensus protocols [9, 10]. Classically, consensus layer protocols only are required to maintain consistency of the blockchain. While additional requirements have been investigated in the past – for example causal order or censorship resilience – very little attention has been given to the fairness of the order of events, making it possible to execute frontrunning or rushing attacks; several such attacks have been observed in the wild already, and there is evidence of bots systematically scanning unscheduled transactions vulnerable to frontrunning [20]. Some blockchains attempt to make such attacks somewhat harder, for example through special protection for the leader, rapid leader change, or a completely leaderless approach, while others can be easily manipulated by a single corrupt validator or a well targeted denial of service attack. In addition to allowing questionable behavior, this can also be a potential regulatory issue, if exchange are required to prevent some levels of fraud.

In this paper, we investigate the issues of attackers manipulating or exploiting the order in which transactions are scheduled in the blockchain. More concretely, we look into *order fairness*, i.e., ways we can assure that the relative order of transactions is fair. We show that one of the more intuitive definitions of fairness is impossible to achieve, and present several alternatives.

Our approach integrates with many existing blockchains without significant change or non-standard assumption on the blockchain implementation . The main requirement is that there is some set of parties (resp. validators) known to each other through which fairness is defined. This comes natural to most voting based protocols, while longest-chain based protocols with an undefined set of participants will need to use a mixed model approach to be compatible with our model. This allows us to combine several variations of fairness with different blockchains, have different degrees of fairness for sets of transactions running on the same blockchain, and even change the configuration on the fly without needing to break the chain. This setup can also come especially handy if one wants to formally verify the protocols – it is vital here to have the small, independently verifiable components and to not need to formally verify dozens of variations of the same protocol.

## 2 MODEL AND ARCHITECTURE

Our model extends the system model and definitions of Cachin, Kursawe, Petzold, and Shoup [6]. Thus, we assume a known set of $n$ validators that are permanently active and can identify each other, and that the number of byzantine corrupted parties is just less than a third of all parties (i.e., $n = 3t + 1$), though the approach can easily be expanded to a staking model or general adversary structures [15]. We assume that an attacker has complete control over the time and order of message delivery, but is not allowed to completely drop a message. Furthermore, we assume that messages are authenticated, and that all participants can sign messages as well as verify each others signatures. In addition to the classical byzantine nodes, we also assume rogue traders might try to game the system to get an unfair advantage, especially to get ahead on performing a transaction. These traders can collaborate with any amount of other traders as well up to a third of the validators; in fact, formally we assume that all traders are under control by the adversary.

In the taxonomy of [13] this would be a asynchronous, permissionless network with point-to-point communication, though we

*AFT '20, October 21–23, 2020, New York, NY, USA*
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8139-0/20/10…$15.00
https://doi.org/10.1145/3419614.3423263

would like to stress that there is no limit on network participation as long as all active validators are aware of each other. For the sake of ease of description, we omit the finer details of a formal model that allows for the clean integration of cryptographic assumptions [6, 13, 14].

One can argue that in modern blockchain implementations, the attack model has gotten somewhat more complex. In addition to validators that are as malicious as possible to violate a security goal (which one usually assumes to be limited to 1/3), there is now also the concept of selfish validators that do not intent to violate a security goal, but to optimize their individual (financial) outcome. For the case of our protocol, this distinction less relevant, as the only safety property we are worried about is fairness; all other safety properties, as well as an infrastructure for incentives, is assumed to be performed externally, e.g., by the underlying blockchain. One consideration in this respect is that it is possible that the incentive mechanism of an underlying blockchain can undermine our protocol by motivating all validators to cheat independently in the same way – if transactions can come with their own fees, and those fees are (at least potentially) distributed to all validators involved in the fairness preprotocol, then every participant has an incentive to prefer high paying transactions; thus, if the protocol is to be used in a setting where this is a danger, the validators performing the fairness preprotocol would require a separate incentive mechanism.

The validators receive external *requests* from the traders. We make no assumption on the timing or the order, which is under complete control from the adversary. The blockchain protocol then *delivers* the requests, i.e., puts them into a block while satisfying the basic properties of atomic broadcast. In practice, to optimize bandwidth, the protocol would likely not use the requests themselves, but hashes thereof. For the sake of presentation, we will use the term request even when a hash would be sufficient. Messages are send by a simple multicast with no requirements on consistency or safety. While there might be some room for optimization if intelligent gossiping protocols are used, our only requirement for the communication layer is that messages between honest parties eventually arrive. An alternative model in the literature is the GST model going back to Dwork, Lynch, and Stockmeier [12], which in some interpretations does allow for some message loss. In this model, the adversary is allowed to arbitrarily delay or drop messages until a time called the *global stabilization time*, after which she needs to deliver all messages within a known timeout. In this model, protocols essentially try to not violate safety before GST, and then assure liveness after. While we don't model our protocols in this setting, they work in it just as well as long as lost messages are resend.

The goal of our design is not to build a new blockchain that includes fairness, but to build a module that can be added to existing blockchain designs. To this end, we provide a pre-protocol that is run by the validators in parallel to the actual blockchain. The preprotocol outputs valid blocks that assure order fairness. While these blocks can be generated by every validator, in most consensus implementations, blocks are proposed by only one or very few parties. To this end, we define a set of designated leader(s) which execute the part of the protocol that generates blocks. The leader part does not involve any communication though, and thus could be executed by every participant without additional communication effort. In addition, we need to modify the block-validity function – proposed blocks are not valid unless it is also verified that the fairness conditions have been satisfied.

To be able to use more established formal definitions, we assume that our protocol communicates with an atomic broadcast subprotocol; for all practical purposes, this is equivalent to a blockchain in our context. We make no assumptions on how the underlying atomic broadcast protocol is implemented, and what – if any – timing assumptions it uses. In fact, our preprotocol can work in a completely different model than the underlying blockchain – while our model has a voting/quorum based approach in mind, the blocks generated by the fairness pre-protocol can es well be processed by a Nakamoto style implementation such as Ethereum or Ouroboros, not unlike the approach that Casper is taking to add finality [5] to a longest chain protocol. We do, thus, assume that the participants in the fairness protocol know and recognize each other, that all participants are trying to deliver a continuous service, and that the underlying incentive model does not create an incentive for all participants to try and undermine fairness. While it would be helpful to assume the same for the blockchain protocol, this is not strictly necessary, and the different models of a longest-chain protocol and the voting based model for our protocol can coexist in parallel, though then for practical reasons with different sets of validators. It is, for example, possible to use the fairness protocols presented here to add order fairness to (some) Ethereum transactions, as long as it is possible to enforce our new validity condition for that chain and assure the the underlying blockchain only accepts the blocks we generate in the order we generated them.

As we envision a blockchain that handles a diversity of transactions, order fairness only needs to be assured for subsets - it is not necessarily required that a request related to trading cryptokitties is treated fairly with respect to a request related to crop prices in Australia. Thus, every transaction has a market-identifier $m_{id}$, and only transactions that have the same market-identifier need to be fair with respect to each other. As we provide different fairness models, it is also possible to use different fairness pre-protocols for different markets, and omit fairness altogether for others. There is even a possibility that a single request has several market identifiers and thus is delivered in a relatively fair way with respect to several, otherwise independent markets. The main issue with the latter is that it adds quite some complexity if we want to have different fairness protocols for different markets. While there is no fundamental issue with this, we do not include this property for our protocols in this paper for the sake of (relative) simplicity.

## 2.1 Related Work

The only work we are aware of that looks at order fairness is parallel work from Kelkhar, Zhang, Goldfeder, and Jules [14]. They also identify the impossibility of strict fairness and resolve to address block fairness. While our approach is to weaken the fairness condition to circumvent the impossibility of block fairness, they define a concept of weak liveness wile maintaining the stronger fairness condition to this end, and define a set of protocols (both synchronous and asynchronous) to provide order block fairness. The price for the stronger fairness is that there is no limit on when requests are delivered or how big a block becomes, though the protocols could easily be adapted to one of our models. Their approach also differs in the architecture - while we aim to have a module to be combined with existing atomic broadcast protocols and provide maximum resilience for asynchronius networks (i.e., n > 3t), their work presents a stand-alone protocol for $n > 4t$.

The concept of *causality* in state machine replication was first introduced by Birman and Reiter [19], with the example of preventing stock trading fraud. The definition was later refined by Cachin, Kursawe, Petzold, and Shoup [6], and again by Duan, Reiter, and Zhang [11]. While the details in the definitions do matter for meaningful proofs and avoiding less straightforward attacks, the basic idea of these definitions is the same; a message is processed by the protocols in a way that its position in the ordering is fixed before any participant learns of its content. While this is sufficient to prevent some financial fraud – especially if we also allow the sender of a request to remain anonymous until the transaction is scheduled – the protection offered by commit and reveal is not sufficient. Especially in cases of high volatility, traders can still get an advantage if they can schedule transactions faster than their peers.

The notion of *fairness* has been used in different contexts in the literature. In the context of block delivery, the concept was formally introduced in [6], though some extend of fairness is already provided by earlier protocols such as Castro and Liskovs BFT protocol [8]. In this definition, fairness essentially requires that a blockchain is fair if the time between $t + 1$ honest parties being aware of a request and that request being delivered is bounded. This concept is somewhat similar (and sometimes used as a synonym) to *censorship resilience* [18], though that term as well has now taken on a multitude of meanings in the literature, and usually does not rule out an unfair delay in delivering a request. In terms of order fairness, fair protocols at least give an upper bound on the level of unfairness – while it is possible that requests are processed in a different order than they arrived, the number of requests that can rush ahead of a particular request is limited. In [17], a different fairness definition is defined – here, fairness requires that all validators get an equal opportunity to get their transactions into the blockchain. This is a different model than we assume, as we want to achieve fairness for transactions coming from external participants, while this protocol assures fairness between the validators. There is some relation though, as fairness between validators assures that the dishonest validators cannot dominate the blockchain, and thus requests seen by all honest validators are processed somewhat fast.

The proof-of-work model has a different approach to fairness. Essentially, if the majority of miners are honest, and the number of transactions is smaller than the maximum the network can handle, the probability that some winning miner will process a given transaction soon is relatively high (though there is no strict upper bound). This effect is diluted by an economic argument though – if (as the case in Ethereum and Bitcoin) it is possible to pay miners for preferred treatment, the delay until a particular request is delivered can become fairly high. In terms of order fairness, this feature makes the blockchains unfair by design – it is explicitly build in that clients who pay more can get preferred treatment. While this form of *capitalistic fairness* makes sense in some settings, it also opens the door for abuse [20], and can lead to prohibitively high fees in busy times.

Some of the more recent protocols [1, 4] frequently exchange the leader even in the absence of observable misbehavior. This makes it harder for an attacker to impose controlled unfairness, as it is harder to assure a corrupted validator is in charge of scheduling when the adversary needs it, though it might be possible to remove the honest leader with a limited denial of service attack. An additional countermeasure is to choose the next leader randomly, decreasing another level of control of the adversary. Fully randomized protocols [6, 18] also make it harder for an attacker to control the level of unfairness. Nevertheless, an attacker can still cause unfairness to a large extent, and – while the unfairness is harder to control – the protocols are not necessarily *order fair*, i.e., preserving the order in which requests come when delivering them.

## 3 THE IMPOSSIBILITY OF FAIRNESS

The term fairness has found numerous definitions in the atomic broadcast and blockchain literature. Most commonly, fairness means one of the following:

- every request eventually gets scheduled
- every request gets scheduled within a bounded time or number of implementation related messages

Additional constraints depend on the model used, e.g., requests only need to be scheduled within a bounded time after GST (Global Stabilisation Time).

For many consensus protocols, fairness does not come naturally. Especially for leader-based protocols, a leader can easily suppress a message. There are a number of countermeasures against this. In [8, 16], replicas watch a leader and dispose of them if they are dishonest; other protocols[1, 2] change the leader frequently, in the hope that an honest leader will eventually handle all outstanding requests. With the exception of [16], no protocol can give strong bounds on when a message is actually scheduled – the time until a message gets scheduled depends on the accuracy of the timing assumptions and is thus depended on an out-of-protocol factor. Leaderless protocols [6, 18] tend to have better implicit fairness protection; while they tend be a little slower than leader based ones (at least in a well-behaved network), the decreased effort to assure fairness can give those protocols an edge in a trading blockchain.

As we are anyhow sorting transactions into blocks (this comes rather natural for a blockchain), though it is possible to use logical blocks that encompass several blockchain blocks. In addition to order fairness, this also assures fairness as defined above. The pre-protocol each party would follow looks as follows (unoptimized version, basing on a leader based atomic broadcast protocol for simplicity):

**Definition 1** (Block Fairness)**.** After a request has been seen by $n-t$ honest parties, it will be scheduled in the next block; if it hasn't been seen by at least one honest party, it will not be scheduled in the next block.

This is relatively easy to implement – before the ordering protocol starts, every validator sends around a list of all requests they have seen; a valid proposal for a block then consists of the transactions out of $n - t$ of these sets that got $t + 1$ votes.

In the setting we envision for our blockchain, even the stronger definition of fairness is insufficient. In addition, we also want *order fairness*, which more captures the intuitive meaning of the word – if one request is send before another request, it would be fair if it is also scheduled first.

**Definition 2** (Order Fairness)**.** A byzantine fault tolerant total ordering protocol is called *order fair* if the following holds: If all honest parties receive request $r_1$ before request $r_2$, then $r_1$ is delivered before $r_2$.

However, this definition of fairness is not only impossible to achieve, but inherently contradictory even if only one party is corrupt.

*Proof (sketch).* Suppose we have $n$ parties $P_1, ..., P_n$, and $n$ requests $r_1, ..., r_n$. Then let party $P_i$ get the transaction requests in the order $r_i, r_{i+1}, r_{i+2}, ..., r_n, r_1, r_2, ..., r_{i-1}$. Now for every $j$, the only party that sees $r_j$ before $r_{j-1}$ is party $P_j$; all other parties see $Pr_{j-1}$ before $r_j$; also, $P_1$ is the only party that sees $r_1$ before $r_n$.

If all parties are honest, then there is no dedicated message order – no two requests will have been seen in the same order by all honest parties. However, if any party $j$ is dishonest, then $r_j$ must be scheduled after $r_{j-1}$, as $P_j$ is the only party to see $r_j$ before $r_{j-1}$ (if $P_1$ is dishonest, $r_n$ must be scheduled before $r_1$).

As the honest parties following the protocol do not know who is dishonest, the outcome of the ordering protocol must be correct independently of which party is dishonest. Thus, for all $i$,$r_i$ must be scheduled before $r_{i+1}$ as well as $r_n$ before $r_1$, which is a contradiction. □

One way out would be to only require $r_2$ and $r_1$ to be in the same block. However, even that might not be possible, and there is another weakness in this definition: The corrupt parties might see $r_2$ long before any honest party would see $r_1$, thus our protocol essentially can't schedule anything seen by $t$ parties only; it seems hardly fair if $t$ validators cannot get a message scheduled that every client can schedule. We leave it to further work to find further definitions for order fairness that are efficiently achievable and might serve some usecases better.

**Definition 3** (Order Fairness, 2. attempt)**.** A byzantine fault tolerant total ordering protocol is called *order fair* if the following holds: If all honest parties receive request $r_1$ before request $r_2$, then $r_1$ is delivered in the same block as $r_2$ or earlier.

This definition is in fact a bit weaker than we require – it is still possible to achieve order fairness as defined before for all transactions that are decidable, and use a separate approach (e.g., a pseudo randomized schedule) for the undecidable ones. As this can easily be added by a local, deterministic algorithm once fairness following the above definition is achieved, and there may be a multitude of ways an application may want to resolve an undecidable situation, we focus on achieving fairness on a block level for Wendy as presented here and leave the block-internal sorting to the implementation.

Unfortunately, we can show that even with the weaker definition we cannot always achieve fairness; more precisely, while there are no undecidability issues anymore, we cannot guarantee termination.

*Proof (sketch).* In above proof, we have shown that there exists a schedule in which the required order of messages depends on which party is faulty, thus requiring to take into account a parameter that is not known to an honest party. In this proof, we build on that construct to design a schedule that would create a block of unlimited size.

For this outline, we assume $n = 4$ and $t = 1$. Consider two schedules as used above, i.e.,
$P_1$: $m_1$, $m_2$, $m_3$,$m_4$
$P_2$: $m_2$, $m_3$, $m_4$,$m_1$
$P_3$: $m_3$, $m_4$, $m_1$,$m_2$
$P_4$: $m_4$, $m_1$, $m_2$,$m_3$

and

$P_4$: $m_5$, $m_6$, $m_7$,$m_8$
$P_3$: $m_6$, $m_7$, $m_8$,$m_5$
$P_2$: $m_7$, $m_8$, $m_5$,$m_6$
$P_1$: $m_8$, $m_5$, $m_6$,$m_7$

Both schedules area split into three segments as shown below:

| | $A_1$ | | | $A_2$ | | | | $A_3$ |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | $m_1$ | $m_2$ | $m_3$ | | | | | $m_4$ |
| $P_2$ | | $m_2$ | $m_3$ | $m_4$ | $m_1$ | | | |
| $P_3$ | | | $m_3$ | $m_4$ | $m_1$ | $m_2$ | | |
| $P_4$ | | | | $m_4$ | $m_1$ | $m_2$ | $m_3$ | |

| | $B_1$ | | | $B_2$ | | | | $B_3$ |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | | | | $m_8$ | $m_5$ | $m_6$ | $m_7$ | |
| $P_2$ | | | $m_7$ | $m_8$ | $m_5$ | $m_6$ | | |
| $P_3$ | | $m_6$ | $m_7$ | $m_8$ | $m_5$ | | | |
| $P_4$ | $m_5$ | $m_6$ | $m_7$ | | | | | $m_8$ |

We now link those two schedules to one combined schedule with the segment order $A_1$, $B_1$, $A_2$, $B_2$, $A_3$, $B_3$.

By the design of schedules A and B, to achieve fairness, $m_1$, $m_2$, $m_3$, and $m_4$ must be in the same block. The same holds for $m_5$, $m_6$, $m_7$, and $m_8$. The argument for this is equivalent to the previous proof; as it is not known to the honest parties who is honest and who not, the requirement could imply that $m_1$ has been seen by all honest parties before $m_2$ (if $P_4$ is corrupt), $m_2$ before $m_3$, $m_3$

before $m_4$, and $m_4$ before $m_1$. Thus, all those messages need to be scheduled in the same block.

In the combined schedule, we also have all honest parties see $m_7$ before $m_4$. Thus, $m_7$ must be scheduled in the same or an earlier block than $m_4$. Similarly, $m_3$ needs to be in the same or an earlier block than $m_8$. As $m_7$ and $m_8$ and respectively $m_3$ and $m_4$ must be in the same block, this means all messages have to be scheduled in the same block.

If we combine the segments the other way around, i.e., $B_1$, $A_1$, $B_2$, $A_2$, $B_3$, $A_3$, we get the same result: $m_7$ is seen by all parties before $m_4$, and $m_3$ is seen by all parties before $m_8$, meaning that still both segments need to be in the same block.

We can now repeat this construction. Suppose we have segment $C$ in the same structure as segment $A$, and segment $D$ in the same structure as segment $B$. Then consider the schedule

$A_1, B_1, A_2, C_1, B_2, A_3, C_2, B_3, C_3$

By above argument, all messages in $A$ and $B$ need to be in the same block; the addition of the messages from segment $C$ does not affect the argument. Similarily, all messages in $B$ and $C$ need to be in the same block; this is unaffected by $A$. In the same way, we can add $D$ in a way that it needs to be in the same segment as $C$:

$A_1, B_1, A_2, C_1, B_2, A_3, D_1, C_2, B_3, D_2, C_3, D_3$

This construction can be arbitrarily repeated, leading to an infinite sequence of messages that all need to be in the same block. □

A notable property of our result is that we do not need a corrupted party to actually act in any bad way – it is enough that there is some party that has the label 'corrupt', and noone knows which one it is. While we haven't worked out the proof, it is likely even impossible if we only require fairness if noone actually is corrupt. To assure liveness in an asynchronous system, the protocol still needs to progress on $n - t$ inputs, which means it misses some information that might be relevant to define a valid order. We did at this point not investigate further, as we prefer to have a protocol that offers somewhat weaker fairness, but maintains robustness in the face of a byzantine adversary.

There are subtle differences in the underlying model that impact what the construction actually means. In some models – essentially the cryptographically sound ones that assume a polynomial time bound adversary [6, 14]- one assumes that the number of incoming (and adversary generated) requests is somehow bounded, i.e., at some point the protocol terminates for good. In this model, our construction does not strictly violate liveness – what happens is that, to satisfy fairness, all requests will be delivered in the one and only block the protocol ever schedules just prior to termination. For those models, we do not prove impossibility of block order fairness, but impossibility of any meaningful efficiency guarantees – in the worst case, order fairness is reached by treating all parties equally bad. If we assume a model that allows for infinite protocol runs, the last point in time does not exist, and a protocol cannot guarantee to deliver anything.

The other interesting modeling aspect is the amount of asynchrony required. In the schedule above, once we start interleaving the D-blocks, all messages in the A-block have been seen by all honest parties. This implies that we do not need a fully asynchronous system. For a consensus between $n$ parties, if $\delta t_r$ is the time interval between the first honest party becoming aware of a request $r$ and the last honest party doing so, then the adversary needs to show honest parties less than $3n$ other requests during $\delta t_r$. Thus, our construction is also possible in most synchronous systems, as long as the adversary can generate/access sufficient requests in the given time-span and has the power to freely determine a schedule in which an honest party sees any set of $3n$ consecutive requests.

Thus, if we bound the number of requests the adversary is allowed to show to honest parties in between the times when the first honest party saw a particular request and the last honest party saw it, the impossibility result still holds.

Theorem 1. *There exists a schedule such that, to achieve block order fairness, all requests any honest party ever seen need to be scheduled in the same block. Consequently, no block can be delivered with this schedule while new requests can be generated.*

*Furthermore, once an honest party has seen a request r, the schedule requires less than* $3n$ *other requests to be operated on until the last honest party sees* $r$.

## 4 CIRCUMVENTING THE IMPOSSIBILITY

We first show a protocol that can guarantee fairness, but does not overcome the liveness issues mentioned above, i.e., it is possible for an adversary to prevent termination. For the ease of description, we describe a somewhat wasteful version of the protocol which resends all requests that did not make it into a block for the next block; in a real implementation, this would be handled in a more efficient way. Also, the protocol as described is sending a lot of signatures repeatedly; that, too, can be optimized in an implementation version.

We describe our protocol as a pre-protocol to the atomic broadcast. The pre-protocol generates a proposal for a block that can then be proposed as the next block for the atomic broadcast protocol, alongside validation information that allow verifying that the block was properly generated. To this end, we assume an atomic broadcast protocol following the definition of [6]. In addition to needing an external validity property, i.e., there is a validation function such that an honest party only accepts an output $r$ with added validation information if the verification function holds. be one party, or every party intending to construct a valid proposal. For simplicity, we also assume that the protocol is re-invoked upon termination by the atomic broadcast protocol, and that the framework assures that messages linked to undelivered requests are replayed to the next incarnation of the pre-protocol in the same order, and messages linked to delivered requests are ignored. The reason to structure the protocol this way (rather than having an infinite loop that invokes the atomic broadcast protocol and taking care of messages itself) lies in the modular architecture we want to allow - the fairness pre-protocol is an optional add-on to the atomic broadcast, and thus should be a pre-protocol invoked by the atomic broadcast rather than the other way around, and it must be possible for one atomic broadcast protocol to use different pre-protocols for different markets.

One issue with this approach is that fairness in the traditional sense – if every instance of the pre-protocol terminates, then every request that is seen by all honest party also is delivered (preferably in a bounded time) in some block – is no longer a property of the pre-protocol, but of the combination. This can however easily be derived from order fairness if we show that every terminated instance of the pre-protocol delivers a non-empty block:

- By assumption, messages that have not been delivered are treated by the next incarnation of the pre-protocol as if they arrived at the same time in the same order
- The protocol guarantees progress, i.e., at least one request is delivered into a block on each terminating incarnation
- By the order fairness requirement, for every request that has been seen by all honest parties, there is a finite number of requests that can be scheduled in an earlier block.

.

We say that a request $r'$ *blocks* another request $r$ given the current information, it cannot be excluded that $r$ needs to be in the same or an earlier block to achieve block order fairness. More precisely, $r'$ *blocks* $r$ if $r$ and $r'$ share a market-identifier, and it is not the case that $t+1$ parties

- have reported to have seen $r$ before $r'$, i.e., assigned it a lower sequence number, or
- have reported to have seen $r$ and all requests with a lower sequence number, but not $r'$.

Lemma 2. *If $r'$ does not block $r$, then $r'$ is not required to be in the same or an earlier block than $r$ by the requirements of block order fairness.*

*Proof.* To be required to be in the same or an earlier block, all honest parties need to have seen $r'$ before $r$. If $t+1$ parties report to have seen $r'$ after $r$, at least one of them is honest, and thus not all honest parties have seen $r'$ before $r$. □

The following defines how a valid vote and block look like:

**Definition 4** (Vote-Validity)**.** A vote is valid if it has the proper format, and once all requests with a lower sequence number from that voter have been received.

**Definition 5** (Block-Validity)**.** A block $\mathcal{B}$ is valid if it contains a nonempty set of requests with $n-t$ valid votes each; a vote for $r$ is valid if it contains the signed votes for all requests for that block with a lower sequence number. Furthermore, for every $r$ in $\mathcal{B}$, if there is a request $r'$ in the vote validation that had at least $t+1$ votes with a lower sequence number than $r$, then $r'$ needs to be in $\mathcal{B}$ accompanied by $n-t$ validation votes.

Theorem 3. *The protocol* Neverending Wendy *guarantees safety, i.e., if a block is sent to the atomic broadcast protocol, and there are requests $r$ and $r'$ such that all honest parties have seen $r'$ before $r$, then $r'$ is in the same or an earlier block than $r$.*

**Pre- Protocol Neverending Wendy for block $b$ and protocol instance $ID$**

**All parties:**

```
let $i$ be the counter of incoming requests, starting at 0.
while no valid proposal has been seen as the proposal for atomic broad-
    cast for block $b$ do
        for all known and unscheduled request $r$, in the order of the
            receiving the requests, send the signed message ($ID$,$b$,$i$,$r$) to
            all parties, where $i$ is the sequence number of that request.
end while
```

**Additional protocol for the leader(s):**

```
$\mathcal{B} = \emptyset$
wait until the first request $r$ is contained in the signed and valid votes
    from $n-t$ parties; add $r$ to $\mathcal{B}$
while any request $r' \notin \mathcal{B}$ blocks any other request $r \in \mathcal{B}$,
        if request $r'$ has at least $n-t$ votes, add $r'$ to $\mathcal{B}$
end while
The proposal for the next block of the atomic broadcast is $\mathcal{B}$, validated
    by all signed votes for requests in $\mathcal{B}$.
```

*Proof.* If the leader is honest, it will place at least one request in $\mathcal{B}$. By the protocol logic, $\mathcal{B}$ will be delivered once no request not in $\mathcal{B}$ blocks any request in $\mathcal{B}$.

As the validity proof contains all the history that lead to the definition of the block, every valid block has to satisfy the conditions for block order fairness. If the leader is dishonest, the only misbehavior (apart from deliberately not terminating the pre-protocol) is to suggest different valid blocks to different parties. This, however, is easily caught by the atomic broadcast protocol. Other dishonest parties can report different orders to different leaders (if those exist). This also is caught by the atomic broadcast protocol (which in this case should select one of those blocks as the next one), as well as requiring contradictory signatures that are then provable exposing the corrupt party.

□

Unfortunately, in a fully asynchronous system, this definition cannot guarantee termination. As we have shown in the previous section, it is possible for an adversary to construct a schedule in which an arbitrary amount of messages needs to be put into the same block; thus, an adversary with sufficient influence on message ordering can keep the protocol process one block forever. Even in a synchronous system, this issue is not automatically resolved. An adversary that needs to respect upper bounds on message delivery can still create a infinite schedule as long as she has full control on the order of message scheduling.

Consequently, we also cannot quantify absolute fairness in this setting – once a request is seen by all honest parties, there is no upper bound on when it is delivered. The only statement we can make is about the block it will be contained in (which depends on the number of undelivered earlier requests), but not on the time or communication effort until that block is delivered. As this issue is tied to the possibility of an infinite loop, the approaches discussed below we are proposing to handle this problem will also allow for absolute fairness, as long as the underlying blockchain can guarantee it.

## 4.1 Armageddon

If the protocol terminates due to lack of usage (i.e., there are no more requests to be scheduled), then the impossibility result no longer holds – in the worst case scenario, the protocol only schedules one block after the genesis block which then contains all transactions (one could argue that such a behavior may hasten the end-of-time scenario as users abandon the system). What is left to show is that all requests that an honest party has seen actually are delivered. This model also assumes that the adversary cannot keep the protocol running forever by generating its own transactions. This would usually be the case as (a) forever is a very long time and a concept that doesn't exit in a cryptographically strict model, (b) usually transactions cost money to incentivise the validators, so such an adversary would spent an unlimited amount of money to prevent protocol termination.

If the protocol terminates while still in operation due to validators opting out. In this case, a weaker form of liveness is required - while the protocol should have created all the blocks it could before, it cannot be expected to deliver every single request in that setting. While we do not quantify which messages can get lost under these conditions, [14] provides the formalism to cleanly define such end-time scenarios.

## 4.2 Relative Synchrony Assumption

One reason why the impossibility result works is that we allow the adversary to completely control the schedule, i.e., the order in which all parties see all requests. This is an unrealistically strong adversary; it is usually defined that way as it is rather hard to model a realistic worst case network attack. In the following, we define an adversary who is almost that strong, but has a (small) failure probability. For this definition, we assume that there is some form of global time, which is unknown to the individual parties.

**Definition 6** (Probabilistic Adversary Failures)**.** If two messages $m_1$ and $m_2$ are sent by honest parties such that $m_1$ has been sent before $m_2$, then with at least probability $p$, $m_1$ is delivered before $m_2$.

While this definition invalidates the impossibility result and allows for an algorithm to achieve order fairness, we still run into practical issues. If $p$ is unknown (analogous to the failure detectors, where it is unknown when a party is rightfully suspected), then we have no known upper bound for the block size and, relatedly, latency. Even if $p$ is known, the maximum possible blocksize can be prohibitively large for any practical implementation.

Note that this definition also adds enough synchrony to allow for deterministic byzantine agreement, as the adversary will (eventually) fail to prevent termination.

## 4.3 Probabilistic Block Order Fairness

**Definition 7** (Probabilistic Block Order Fairness)**.** A byzantine fault tolerant total ordering protocol is called *probabilistically block order fair* if the following holds: There is a fixed probability $p$ such that, if all honest parties receive request $r_1$ before request $r_2$, then $r_1$ is delivered in the same block as $r_2$ or earlier for with at least probability $p$.

This definition allows a protocol to at some point stop assuring fairness and put the already processed messages into the next block, even if that means that some messages are scheduled unfairly. To achieve termination at sacrificing some level of fairness, we can set a threshold $r_{max}$ and artificially terminate the protocol once the number of requests in $R$ exceeds $r_{max}$. This means that an adversary with sufficient network control can cause a limited amount of unfairness (i.e. scheduling some requests out of a fair order), however, the majority of all requests will be scheduled fairly, and causing an unfair order does require a very high level of network control for the adversary. Of course, the cut-off point can also be defined taking other factors into account, e.g., a timeout, the number of requests in the queue, etc.

We can strengthen this approach by adding a random factor. In that setting, once $r_{max}$ is exceeded, we use a common coin [7] to determine when the protocol stops. This could be done in a way that the result is unpredictable even for the leader - after each request added to $B$ beyond $r_{max}$, the leader can request a coin from all other parties defining on whether or not she should stop at that point. Thus while an adversary with extensive network control can cause an unfair scheduling, she has no influence on who is treated unfairly. Communication overhead can be managed by piggybacking the coin shares to the voting messages; furthermore, as the attacker gains little apart from a small slowdown of the protocol, one could hope that most economic attackers would not attempt such an attack, and thus in most cases the protocol terminates before reaching $r_{max}$. While this allows the timing model to be unchanged, the required maximum blocksize is linked to $p$; if $p$ is to be very small (e.g., one in a million), the number of messages that in a block that the protocol needs to be capable of handling is correspondingly high.

One blemish with this approach (as well as the definition) is that the probability distribution is somewhat ugly. Ideally, one would like an equal probability for each transaction in a block under attack to be treated unfairly; out approach can not deliver that, but is aimed to create enough uncertaincy that a targetet attack on a specific transaction will have a significant likelyhood to fail. Given the dependency on the attacker behaviour, it is not clear at this point if a better definition can be made; this would be an interesting topic for future work.

One important point to note is that while our protocol only provided block fairness, in most cases, it is possible to provide transaction-level fairness as defined in Definition 2, as the the loops that cause an undecidable schedule should be reasonable scarce during a normal run. While Wendy as described here only assures Block order fairness, it also attaches sufficient auxiliary information to the transactions to allow a (deterministic) post-processing algorithm to make a best effort in sorting the transactions inside a block, i.e., provide order fairness where possible and fall back on a backup mechanism (e.g., a pseudorandom schedule) for those blocks where such an order does not exist. Thus, in spite of our impossibility proofs, the protocol as presented so far can be expected to provide order fairness for most transactions during normal conditions, even if our formal model allows for an adversary who assures it never happens.

It is straightforward to integrate such a function into Wendy, and have transactions inside each block sorted as fair as possible before a block is handed over to the blockchain. However, as we also see cases where a higher level application may want a say in fairness conflict resolution, we omit this here.

## 4.4 Fairness using Local Clocks

We now present a different definition of fairness that is slightly weaker, but that allows for much stronger liveness guarantees.

**Definition 8** (Timed Order Fairness)**.** Suppose that all parties have access to a local clock. If there is a time $\tau$ such that all honest parties saw (according to their local clock) request $r$ before $\tau$ and request $r'$ after $\tau$, then $r$ must be scheduled before $r'$.

Note that there is no need for the local clocks to be synchronized at all; the only formal requirement is that the clock always counts forward and that no two timestamps are the same. Obviously, the definition does make more practical sense however if the clocks are roughly in sync. Using GPS as a timesource with a hardening layer to prevent GPS spoofing (e.g., [3]) should be sufficient to make this approach practical. It is also possible to emulate clocks by a counter of transactions the party has seen. This counter does have the required properties for our clock, and also can be assumed to be reasonably synchronised between honest parties. This also can be combined with blocktime of the underlying blockchain, i.e., the blocktime corresponds to minutes and the number of (not yet scheduled) transactions corresponds to seconds. This approach would make it easier for new parties to join and resynchronise the logical clocks if something went wrong.

For our protocol, it is sufficient to assure that if $r$ needs to be scheduled before $r'$, $r$ is in an earlier or the same block. As the timestamps are included in the block, ordering of requests inside a block can be performed localy after the block is delivered.

Since the fairness condition changed, the validity of a vote and of a block also look different.

**Definition 9** (Timestamped Vote-Validity)**.** A vote is valid if it has the proper format, and if the sequence number matches the sequence on timestamps on requests from that party. Once a party mismatches the timestamps and the sequence numbers, i.e., there are two requests $r_1$ and $r_2$ such that $r_1$ has a lower sequence number and a higher timestamp than $r_2$, this and all following votes from that party are considered invalid. Furthermore, a vote is only considered valid once all requests with a lower sequence number from that voter have been received.

**Definition 10** (Timestamped Block-Validity)**.** A block $B$ is valid if it contains a nonempty set of requests with $n-t$ valid votes each; a vote for $r$ is valid if it contains the signed votes for all requests for that block with a lower sequence number. Furthermore, for every $r$ in $\mathcal{B}$, if there is a request $r'$ in the vote validation that had $t+1$ votes with a lower sequence number than $r$, then $r'$ needs to be in $\mathcal{B}$ accompanied by $n-t$ validation votes.

```
                    Clocked Wendy for instance ID
                            All parties:
let $i$ be a counter for incoming requests, starting at 0
while no valid proposal has been seen as the proposal for atomic broadcast for block $b$ do
        for all known and unscheduled requests $\hat{r}$, in the order of the timestamps on the requests, send the message ($ID$,$b$,$i$, timestamp($\hat{r}$),$\hat{r}$) to all parties, where $i$ is the sequence number of that request.
end while
                  Additional protocol for the leader(s):
$\mathcal{B} = \emptyset$
wait until the first requests $r$ is contained in the signed list of $n-t$ validators; add $r$ to $\mathcal{B}$
let $\mathcal{R}$ be the set of requests for which a vote for with a smaller timestamp than $r$ was received
        wait until there is a set of $n-t$ parties from which valid votes for all requests in $\mathcal{R}$ are received
    for all $r' \in \mathcal{R}$, if timestamps of $t+1$ votes are smaller for $r'$ than the median of the timestamp of the votes for $r$, add $r'$ to $\mathcal{B}$
The proposal for the next block of the atomic broadcast is $\mathcal{B}$, validated be the corresponding signed votes in $\mathcal{B}$
```

Theorem 4. *(Safety) If a request $r$ is scheduled in a block $\mathcal{B}$, and there is a request $r'$ such that there is a time $\tau$ in a way that all honest parties saw $r'$ before $\tau$ and $r$ after $\tau$, then $r'$ is in $\mathcal{B}$ or an earlier block.*

*Proof.*

Assume without loss of generality that every timestamp has a unique time. This can easily be assured locally by a high enough time resolution, and by ordering votes by party identifyer if two votes have the exact same timestamp.

Suppose at the end of the pre-protocol, we have request $r' \in \mathcal{B}$ and $l \notin \mathcal{B}$, and that $l$ has not been scheduled in an earlier block. Let $\tau_1$ be the median of the timestamps of $r$.

(1) As $r' \in \mathcal{B}$, at least $t+1$ parties timestamped $r'$ before or during $\tau_1$
(2) As $l \notin \mathcal{B}$, at most $t$ parties timestamped $l$ before $\tau_1$.

Suppose by the requirements of timed order fairness, we have to schedule $l$ before $r'$. As $t+1$ of the parties that issued votes are honest, this implies that

(3) there exists $\tau_2$ such that $t+1$ votes contain timestamps for $l$ before $\tau_2$, and at most $t$ votes contain timestamps for $r'$ before $\tau_2$.

By (2), at most $t$ timestamps for $l$ are smaller than $\tau_1$, and by (3) at least $t+1$ are smaller than $\tau_2$; thus, $\tau_1$ is smaller than $\tau_2$. Similarly, for $r'$, by (3) at most $t$ timestamps are smaller than $\tau_2$, by (1) and at least $t+1$ are smaller or equal to $\tau_1$. Thus, $\tau_2$ is smaller than $\tau_1$. This is a contradiction, and therefore it is not possible that $l$ needs to be scheduled before $r'$. □

Theorem 5. *If some honest party sees some request, any honest leader will terminate the protocol with a proposal.*

*Proof.* As every party sends every request it sees for the first time to all other parties, every request that is seen by some honest party is seen – and send to the leader(s) – by all honest parties. Thus, there is some $r$ that is in the signed list of $n - t$ parties. Once a leader gets $n - t$ votes for some $r$ for the first time, there is a finite number of requests $r'$ for which the leader received a vote before. As the leader has seen this vote and is honest, it also forwarded the $r'$ to all other parties, and thus will receive $n - t$ votes eventually. Therefore, the waiting statement always terminates for all requests $r'$. □

Note: We only need successful termination if an honest leader exists. All atomic broadcast protocols we are aware of either have a single leader which is replaced if a liveness problem occurs, or use more than $t$ parties in a leader-like function simultaneously and thus guarantee that there is some honest leader.

## 4.5 Optimizations

The two protocols described above can also be combined. The joint protocol would act like the neverending protocol up until $r_{max}$; however, instead of aborting the protocol and allowing for plain unfairness, it switches to the weaker timed definition of fairness once $r_{max}$ is exceeded. That approach allows for much more aggressive thresholds, as the fallback protocol is no longer unfair, but still fair with a slightly weaker definition.

*4.5.1 Latency and performance impact.* Introducing any kind of order fairness always has a latency impact. If no fairness is required, every incoming request can be processed as soon as it arrives. Order fairness, no matter how it is defined, requires to wait if there might be other requests with a higher priority. While the *mostly fair* protocol allows to parameterise the trade-off between latency and unfairness – the lower the cutoff parameter, the faster the worst case protocol and the easier for an adversary cause an unfair transaction. However, in the benign case, the latency overhead should be reasonably small.

One (small) speedup can be reached by parallelizing the leader part of the protocols. Instead of waiting for the first request to add to $\mathcal{B}$ and then sticking to it, the protocol can be run in parallel for all requests that have been reported by enough parties. In that case, the first instance that terminates its **while** condition wins and defines the next block. It is also possible to cut the threshold in the neverending fairness protocol to $t+1$ by using a more sophisticated blocking function.

Another parallelization approach would be that the first part of the protocol where all parties broadcast their orders is permanently performed, independently of the state of the second phase or the atomic broadcast. Thus, in most cases, once the atomic broadcast starts processing the next block, enough votes should have arrived to terminate the pre-protocol quite rapidly. This approach also has an interesting impact on the overall architecture – rather than having a simple API to call the pre-protocol, some part of it now needs to permanently run in the background. Alternatively, to save overhead, this could also be included as a piggyback in the gossiping protocol.

An additional approach to optimize the Neverending protocol is to allow requests to be removed from $\mathcal{B}$ again. Recall that a request is added to $\mathcal{B}$ if it has received $n - t$ votes and still blocks a request already in $\mathcal{B}$. This is necessary as we can no longer rely on getting more votes concerning this request, and to guarantee progress, this request now needs to be treated as if we know that is has to be in the same block as the one it blocks. However, as additional votes come in, it is possible that it unblocks again. In this case, $r$ and all requests that where added to $\mathcal{B}$ due to blocking $r$ can be removed from $\mathcal{B}$ again, potentially releasing the block earlier.

For the timed protocol, a similar approach can be taken. For this protocol, we have the advantage that for each request $r$, there is a finite number of requests that are blocking it. This blockage is released either once the corresponding request has $t+1$ timestamps smaller than the median timestamp on $r$ (in which case we know if any other request needs to be scheduled before $r'$, it also needs to be scheduled before $r$), or if it got $n - t$ timestamps of which at most $t$ are smaller than the median of $r$ (in which case it can and will be scheduled after $r$). To fully optimize latency, we also need constantly verify if new incoming votes increase the median of a subset of $n - t$ votes for $r$, as a higher median increases the possibility that another request can be decided before it got $n - t$ votes.

With this modification, we believe that the protocols have optimal latency within our modular architecture, i.e., it is not possible to hand a block over to the atomic broadcast protocol earlier. The (informal) argument for the block fairness protocol goes as follows (from the point of view of a leader):

- Every request that $r$ got $t + 1$ votes gets its own $\mathcal{B}_r$, i.e., a potential block containing $r$ and all other requests that have to be in the same block as $r$.By our fairness condition, we cannot deliver any request that has seen less than $t+1$ votes, as it is possible that another request that is unknown at this point has $n - t$ votes that prioritise it over $r$ and thus has to be in the same block. Therefore, for every request that can be in the next block, the protocol maintains has a $\mathcal{B}_r$
- At any point in time, $\mathcal{B}_r$ is minimal; the only requests in $\mathcal{B}_r$ are requests that either have to be in the same block as $r$, or might have to according to the information available.
- $\mathcal{B}_r$ cannot be finalized while it contains a request $r_1$ that is blocked by another request $r_2$ with less than $t + 1$ votes, as $r_2$ might still be blocked by a yet unseen request. Thus, the protocol finalizes $\mathcal{B}_r$ at the earliest possible occasion.

A similar argument holds for the timed protocol; again, the protocol maintains a separate $\mathcal{B}'_r$ for all eligible processes, and decides about all other requests at the earliest opportunity – either once it is clear that they can to be in the same block, or once enough votes are seen to conclude they don't need to.

If we further want to optimize latency, we could open up the modularity of our approach. Most voting based atomic broadcast protocols start with the leader(s) broadcasting the content of the next block (or a hash thereof). Due to the pre-protocol, we already know that $n-t$ parties have seen the content of the requests in that block. Optimizing the interplay between the fairness pre-protocol, the atomic broadcast, and the underlying gossip/multicast protocol is thus certainly promising, but out of the scope of this paper. It also is possible to integrate our protocol deeper with the blockchain implementation. With some modifications it could, for example, replace the first phase of the ABC protocol from Cachin, Kursawe, Petzold and Shoup [6]. As our goal is a modular approach though, we will not follow that path at this point.

*4.5.2 The combined protocol.* There is a set $\mathcal{D}$ of transactions that are ready for the atomic broadcast layer to use. For the ease of presentation, we assume that the communication layer is aware of $\mathcal{D}$, and omits any voting messages associated to any transaction in $\mathcal{D}$. Furthermore, there is a queue $Q$ with which the protocol communicates with the atomic broadcast. The atomic broadcast protocol takes the requests in $Q$ from one or several leaders, adds a block to the blockchain, and then deletes the scheduled requests from the queues from all leaders.

This version of the protocol is defined as a permanent service that takes in requests, and outputs blocks for the atomic broadcast protocol.

## 4.6 Fairness and Advanced Staking

While the protocol described above is relatively model-independent, it is described in the classical committee model, i.e., we have $n$ parties with one vote each, up to $t, n \geq 3t+1$ can suffer from byzantine corruptions. This model translates easily into a stake-based model, where voting power is related to the stake parties have. To allow our results to be applicable for more different staking models, we consider the hybrid-adversary-structure model [15]. In short, this model generalizes the model by replacing the thresholds by the corresponding properties that are required to perform the proof; for example, the threshold $t+1$ is replaced by *sets of parties of which at least one is honest*, while $n-t$ corresponds to *the largest sets of parties we can afford to wait for without having to rely on potentially corrupt parties.* This allows to not only model weighted votes, but also take into account properties, e.g., requiring more 2/3 of the stake in more than 2/3 of a set of defined geographic regions to be honest. In the proofs for our protocols, the two aforementioned properties are the only properties we need, and the proofs can be generalized in a straightforward way. Thus, any staking model that can be formulated this way is compatible with the order-fairness protocols.

## 5 PRACTICAL CONSIDERATIONS

While our approach fits nicely into most voting based protocols due to a similar communication structure, integration with longest-chain style protocols that can have a more dynamic validator structure is not as straightforward – even our definitions of fairness do not make much sense in a Bitcoin or Ethereum style setting where validators don't need to be aware of each other and have no requirement to provide a continuous service. For those protocols, the fairness protocol would need a set of validators independent of the actual blockchain, as has been done with Casper, the finality add-on for Ethereum [5]. Since (just as finality in the case of

**Pre- Protocol Hybrid Wendy for block $b$ and protocol instance $ID$**

**All parties:**

```
let $i$ be the counter of incoming requests, starting at 0.
while true do
    for all first seen and unscheduled requests $\hat{r}$, in the order of
        the timestamps on the requests, send the message $(ID,b,i,$
        $\text{timestamp}(\hat{r}),\hat{r})$ to all parties, where $i$ is the sequence number
        of that request.
end while
```

**Additional protocol for the leader(s):**

```
while true do
    once a request $r$ is contained in the signed and valid votes from
        $t+1$ parties, set $\mathcal{B}_r$ to $\{r\}$
    while for any $\mathcal{B}_r \neq \emptyset$ any request $r' \notin \mathcal{B}_r$ blocks a request
        $r \in \mathcal{B}_r$ and there is no $\mathcal{B}_x$ of order $> r_{max}$
            if request $r'$ has at least $t+1$ votes, add $r'$ to $\mathcal{B}_r$
            if a request $r' \in \mathcal{B}_r, r' \neq r$, no longer blocks any other
                request in $\mathcal{B}_r$, remove $r'$ from $\mathcal{B}_r$
    end while
    for all $r$ for which no request in $\mathcal{B}_r$ is blocked by a request
        $r' \notin \mathcal{B}_r$,
            add $\mathcal{B}_r$ to the $Q$, validated by all signed votes for requests
                in $\mathcal{B}_r$.
            add all $r' \in \mathcal{B}_r$ to $\mathcal{D}$, and remove them from all sets $\mathcal{B}_x$
    if there is a $\mathcal{B}_x$ of order $> r_{max}$
            set all $\mathcal{B}_x = \emptyset$
            while all $\mathcal{B}'_x = \emptyset$
                    for all requests $r$ contained in the signed list of
                        $n-t$ validators, set $\mathcal{B}'_r = \{r\}$
            for all $r$ relating to a nonempty $\mathcal{B}'_r$,
                    let $\mathcal{R}_r$ be the set of requests for which a vote for
                        with a smaller timestamp than $r$ was received
                    let $m_r$ be the largest median of any set of $n-t$
                        votes received for $r$
                    once for all requests in $\mathcal{R}_r$ $n-t$ valid votes or
                        $t+1$ votes with timestamps smaller than $m_r$
                        are received,
                            for all $r' \in \mathcal{R}_r$, if timestamps of $t+1$ votes
                                are smaller for $r'$ than $m_r$, add $r'$ to
                                $\mathcal{B}_r$
            add $\mathcal{B}'_r$ to $Q$, validated by all signed votes for requests in
                $\mathcal{B}'_r$.
            add all $r' \in \mathcal{B}'_r$ to $\mathcal{D}$, and remove them from all $\mathcal{B}'_x$
    end if
end while
```

Casper) order fairness is not required for all transactions on the blockchain, but rather a set of transactions for a specific usecase or market that relate to each other (and potentially a specific smart contract), different sets of additional fairness validators could be used for different applications. Another consideration is that our protocols – as presented here – cannot handle a blockchain that does not provide finality. The necessary changes are not overly complicated. Essentially, we'd need to add the ability to rewind the protocol if a block it assumed delivered is undone.

Our model makes few assumptions on the limits of an attacker - as long as messages arrive, we assume the attacker has complete control over the network. While this is (hopefully) unrealistically strong, it is also necessary – given the multitude of network attacks an advanced attacker can make, any constraints on their abilities should be done with great care, especially in a network that is supposed to have a very long lifetime. However, this means that few of the worst case scenarios can be expected to be seen in reality. Given a block post-processing best-effort algorithm to fairly sort transactions inside a block, we would expect most transactions to even satisfy order-fairness on a transaction level. As we have not tested the protocol on a realistic network yet, this remains a conjecture for now.

## 5.1 Integration Interfaces

The goal of our design is to keep Wendy as modular and easy to integrate into an existing blockchain as feasible. There are three necessary points of interaction with the underlying blockchain, as well as two optional ones.

**Communication Infrastructure (optional)** Unless the fairness protocol is required to provide its own set of validators, we assume that we can use the communication and authentication infrastructure provided by the host blockchain. More precisely, we require access to a multicast primitive that sends messages to all other validators, as well as the possibility to identify messages from other validators as well as create and verify all necessary signatures. While we do not require the multicast primitive to have any security properties, as common in voting based protocols we do assume eventual message delivery, i.e., messages are resend until they arrive or have become irrelevant. While this is not strictly necessary – an implementation of the fairness protocol could re-implement this infrastructure – it would be extremely helpful to have interfaces to those primitives. For a staking or adversary structure based system, we also require access to the appropriate evaluation functions to identify the equivalent of $t + 1$ and $n - t$ votes.

**Block Processing (Optional)** In addition, we may require a block processing functionality. On the protocol level, if fairness requires a request $r$ to precede another request $r'$, we only assure that $r$ and $r'$ end up in the same block. Fine grained ordering – where possible – then needs to be performed separately. As this would usually done by a deterministic algorithm during block post-processing, we do not detail this part in this paper; depending on the fairness definition we used, this algorithm can make a best effort to maintain fairness where possible given the information provided to it by the pre-protocol, which should be straightforward. It is an open (and application-dependent) question at which point in the protocol this functionality should be executed. One possibility is to make this a part of Wendy, i.e., Wendy already proposed internally fully sorted blocks. The advantage of this approach is that this would eliminate an additional interface to the host system. On the other side, given the impossibility results which lead to the possibility of fairness compromises, it might be helpful for a higher level application to have some decision in the choice of the conflict resolution mechanism.

**Block Validity** The validation of a correct block needs to take into account the new fairness rules, i.e., verify the signatures generated by Wendy and check that the block was generated following the rules, and assure that the order of sets proposed by Wendy does not get changed, i.e., blocks relating to the same market-identifyer are proposed in the order Wendy schedules them. While this is easy to implement, it does require a change on the side of the blockchain implementation. The validity function would also implicitly handles the fee-based incentive of underlying blockchains such as Ethereum – as an out-of-order block will be considered invalid, there is no motivation for a validator to propose the blocks in the wrong order, even if the later one would offer a higher fee.

**Transaction Intercept** : Rather than going straight into the local pool of unfinished transactions, all incoming transactions with a fairness requirement need to be handed to the Wendy first. Transactions that have no fairness-label are passed through right away into the pool. The other transactions are processed by the pre-protocol and handed to the pool as a fairness-block.

**Chain access** : Wendy needs to know which transactions have been scheduled in a chain, and – in case of a blockchain that does not offer finality – sufficient information to rewind and get the new state of the chain if a fork is resolved.

The main change on the blockchain side is that it now needs to handle blocks of transactions in addition to individual ones, both with respect to block creation and the underlying incentive scheme. While there are several ways to implement this - e.g., disguising the block of transactions as one big transaction. In addition, the scheduling algorithm should assure that Wendy-generated blocks are not proposed out of order - while this is verified in the block validation, proposing invalid blocks has a rather bad impact on performance. One additional challenge is posed if the size of the blocks generated by the fairness protocol can be larger than the blocks processed by the underlying blockchain. In this case, a mechanism is required to cut a fairness-block into several blockchain blocks, and reassemble them after they have been scheduled by the blockchain. While this is possible, implementing such a functionality on a blockchain with frequently changing leaders is not trivial. For this paper, we assume that the underlying blockchain can process sufficiently large blocks, and instead propose mechanisms to assure that the fairness blocks stay reasonably small.

## 5.2 Performance Estimates

Not taking blocking into account, the additional overhead corresponds to one multicast, i.e., the time $n - t$ honest parties require to send a simple message to the leader (assuming a leader-based protocol). This can be done in parallel to the actual protocol; thus, in a protocol that does not process blocks in parallel (such as Tendermint or HotStuff), the protocol adds to bandwidth usage, but – not taking into account blocked messages – adds almost no latency on a well used network (in fact, block generation should suffer no latency impact at all, apart from the effect of a slightly higher network utilization; the way latency comes in is that individual transactions can end up in a later block than they would without fairness. The main latency impact thus is through messages being blocked through our fairness rules. If the network is reasonably well behaved, and the message ordering is not optimized to slow down the protocol, a transaction blocking another should be resolved rather fast. If a transaction is blocked for an honest leader, it will take at most two message round-trips until that leader receives enough information to unblock the message. This is still well under the block processing time of most blockchains, which would mean that most transactions will end up in the same block as they would without fairness, and the remaining ones are very likely to end up in the next block.

The impact of the communication overhead depends on the underlying blockchain. In an extreme case, if a longest chain protocol with a large amount of validators is used that would all participate in Wendy, the overhead would be massive. For a voting based protocol, the overhead corresponds to one step of the protocol, and thus imposes a comparatively low communication overhead. This holds especially as it is possible to use hashes of transactions rather than the full content, which makes the signatures the dominant part of communication complexity. However, as the performance impact generated here depends highly on the actual network behavior, firm statements on real world behavior require running experiments or simulations.

One factor that has a big impact on real world performance is the usage of market-identifiers. Transactions that have no fairness requirements will not be delayed at all, and transactions can only be blocked by transactions with the same market-identifier, which means that – at least for our envisioned usecases – most transactions on a blockchain will not affect each other.

## 6 CONCLUSION

We have shown that order fairness is one of the many desirable properties that is impossible to achieve in a byzantine fault tolerant setting. We have mitigated this by providing slightly weaker definitions of what fair is. We have a presented several protocols to achieve order fairness with these definitions, as well as a hybrid version that can switch between two levels of fairness to avoid the impossibility result. Our protocols are largely blockchain agnostic (with some additional work needed to combine them with longest-chain based protocols), and can be added to almost any protocol that provides a known and active set of validators. Furthermore, our protocols have optimal resiliency in the asynchronous model (i.e., $n \geq 3t + 1$) and optimal latency in terms of message passing rounds within our architectural model.

## REFERENCES

[1] Abraham, I., Gueta, G., and Malkhi, D. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR abs/1803.05069* (2018).

[2] Amoussou-Guenou, Y., Pozzo, A. D., Potop-Butucaru, M., and Tucci Piergiovanni, S. Dissecting tendermint. In *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers* (2019), M. F. Atig and A. A. Schwarzmann, Eds., vol. 11704 of *Lecture Notes in Computer Science*, Springer, pp. 166–182.

[3] Bondavalli, A., Brancati, F., and Ceccarelli, A. Safe estimation of time uncertainty of local clocks. pp. 1 – 6.

[4] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on BFT consensus. *CoRR abs/1807.04938* (2018).

[5] Buterin, V., and Griffith, V. Casper the friendly finality gadget. *CoRR abs/1710.09437* (2017).

[6] Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings* (2001), J. Kilian, Ed., vol. 2139 of *Lecture Notes in Computer Science*, Springer, pp. 524–541.

[7] Cachin, C., Kursawe, K., and Shoup, V. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology 18*, 3 (2005), 219–246.

[8] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999* (1999), M. I. Seltzer and P. J. Leach, Eds., USENIX Association, pp. 173–186.

[9] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., and Juels, A. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *CoRR abs/1904.05234* (2019).

[10] Danezis, G., Hrycyszyn, D., Mannering, B., Rudolph, T., and Šiška, D. Vega protocol: A liquidity incentivising trading protocol for smart financial products.

[11] Duan, S., Reiter, M. K., and Zhang, H. Secure causal atomic broadcast, revisited. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017), pp. 61–72.

[12] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (Apr. 1988), 288–323.

[13] Garay, J., and Kiayias, A. Sok: A consensus taxonomy in the blockchain era. Cryptology ePrint Archive, Report 2018/754, 2018. https://eprint.iacr.org/2018/754.

[14] Kelkar, M., Zhang, F., Goldfeder, S., and Juels, A. Order-fairness for byzantine consensus. Springer-Verlag.

[15] Kursawe, K., and Freiling, F. C. Byzantine fault tolerance on general hybrid adversary structures. Tech. rep., RWTH Aachen, 2005.

[16] Kursawe, K., and Shoup, V. Optimistic asynchronous atomic broadcast. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings* (2005), L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds., vol. 3580 of *Lecture Notes in Computer Science*, Springer, pp. 204–215.

[17] Lev-Ari, K., Spiegelman, A., Keidar, I., and Malkhi, D. Fairledger: A fair blockchain protocol for financial institutions. *CoRR abs/1906.03819* (2019).

[18] Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 31–42.

[19] Reiter, M. K., and Birman, K. P. How to securely replicate services. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 986–1009.

[20] Robinson, D., and Konstantopoulos, G. Ethereum is a dark forest. https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff, 2020. Accessed: 2020-09-09.