Wendy grows up

Klaus Kursawe (klaus@vega.xyz)
Version 0.19

October 28, 2020

1 Introduction

In recent years, blockchain applications have increased in complexity and utility, allowing more advanced financial tools such as exchanges and trading markets to be decentralised. The introduction of decentralised trading markets highlights a number of new challenges for consensus protocols [1, 2]. Classically, consensus layer protocols are only 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. 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 exchanges are required to prevent some levels of fraud.

In a previous paper [4], we introduced Wendy, a pre-protocol to blockchains that can assure fairness. We now present further development of Wendy.

- The protocol is now divided into a framework and the fairness enforcement. The definition of fairness, as well as some assumptions on the adversary strength, are affecting only the fairness enforcement part. Thus, the framework can work with several different definitions of fairness, and easily switch between them as well as using different definitions of fairness for different applications or markets on the same blockchain. This also makes it easier to prove that a fairness definition works.
- We have built an implementation of Wendy running on a simulated network and blockchain. This now allows us to provide first numbers on the

performance overhead caused by Wendy, and can be used to test different fairness definitions.

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 naturally 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.

One goal of this paper and the accompanying implementation is to provide a base for discussion and further development of the protocol before the final implementations(s). The protocol presented here is already much more implementation friendly and adaptable than the original [4]. However, there still may be new insights on integration with existing blockchains, special requirements to work with longest chain based protocols, and definitions of fairness or new use cases that require tweaking of the framework protocol to work properly.

2 Model

The Wendy protocol consists of two parts, the framework protocol and the fairness enforcement. This allows us to support several different concepts of fairness relatively easily – the framework stays the same, and only the enforcement part needs to be replaced. It also allows support of different fairness definitions in parallel – different classes of transactions can be relative fair to each other independently, and with different definitions, but use the same framework protocol. The framework and the fairness enforcement also can have slightly different models.

2.1 The Framework model

The framework itself has quite few requirements; it has no timing requirements and no requirements on the number of corrupt parties. We assume some form of multicast (or gossip) protocol, where one party can send a message to all others. This multicast does not require consistency or safety, though the protocol can be made more efficient if those properties are provided. For this description, we assume that messages between honest parties might be arbitrarily delayed, but will eventually arrive; it is possible to relax this assumption to some extent (see Section 6.3), though for the ease of description we will do that separately.

We also assume that the set of parties/validators is known, and that a mechanism for verifying message authenticity and signatures is in place.

The protocol we present is a pre-protocol that can run before or in parallel to the actual blockchain. Rather than sending transactions to the blockchain for scheduling, they are intercepted by Wendy, processed, and eventually delivered to (or scheduled by) the blockchain. To this end, Wendy creates blocks of transactions that need to be scheduled together, which can either be (part of) a block in the original chain, or abstract into virtual blocks that could also be larger than the blocks of the underlying chain.

To avoid a bottleneck through assuring a fair relation between unrelated transactions which don't need fair ordering (say, two transactions related to derivatives on the price of sport cars in Paraguay and the weather in Tokyo, respectively), we sort the transactions into fairness groups in a way that only transactions in the same group need to be fairly sequenced with respect to each other. To this end, we assume that transactions have *labels* indicated which group they are in. This label can separate applications (i.e., a smart contract can have its own label), as well as application internal streams (e.g., different markets on a trading platform). While the protocol sorts all transactions, only transactions with the same label influence each other. In practice, the labels would be defined by market makers, application providers, or the smart contract.

Wendy per se only cares about *block order fairness*, i.e., if a transaction tx_1 is required to be scheduled before a transaction tx_2, Wendy only assures tx_2 does not end up in an earlier block than tx_1. This was originally an effect from an impossibility result on the most straightforward fairness definition, where schedules exist that cannot guarantee more than block order fairness [3, 4]. Wendy does, though, supply each block of transactions with sufficient additional information that a (deterministic) inner-block order algorithm can also enforce fairness conditions between transactions in one block.

An aditional point in defining fairness protocols is that the classical Byzantine model may need a reinterpretation to take into account the game theoretical aspects. As every validator has a financial incentive to cheat, as well as (potentially) an easy and hard to detect way to do so, we should assume that every validator is trying to subvert fairness, while only a threshold (e.g., a third) do so in a coordinated way.

2.2 Fairness model

Individual fairness definitions can (and probably have to) imply additional constraints, such as limiting the number of allowed corruptions. It is also, for example, completely plausible that a blocking function makes its own timing assumption, (e.g., that all votes that could block a transaction arrive within 5 seconds of the transaction, and thus any transaction older than 5 seconds is considered to be no longer blocked by potentially unknown transactions). The same way the blockedBy function could ignore missing votes missing for more than 5 seconds and simply unblock. This would allow a fairness policy to allow for message loss at the price of making a synchrony assumption, which seems a fair tradeoff for some applications.

2.2.1 Selfish Validators vs Byzantine Validators

In the context of fairness, the original byzantine corruption model – assuming that a fraction of validators are collaborating with (part of) the network to prevent the protocol from achieving its goal by any means – is comming to its limits. While we still need to consider this kind of attack, there's also a weaker version of selfish validators. These validators have no interest in some attacks (e.g., to prevent termination of the protocol), will be unlikely to launch advnaced attacks that require a great deal of ressources, shy away from attacks that can be detected and punished afterwards, and generally are less organisedthan their Byzantine counterpart. They are opportunistic though, and if they see an easy way to undetectedly insert a transaction to that financially benefits them, they would jump at the opportunity. While this attacker is substantially weaker, we can assume that more validators are tempted to be selfish than could realistically be mustered to join a global cabal to bring down our protocols; idealy, a protocol should tolerate all validators to behave in a selfish way. This does come rather naturally in most cases. Protocols that can withstand, say, 1/3 of Byzantine, coordinated corruptions usually can sustain many more uncoordinated once. We do not have a good formal model for this combination yet though, as there are a number of interesting challenges:

- Interaction between selfish and byzantine. If we simply add selfish validators to the Byzantine model, the byzantine adversary could monitor the selfish validators, and jump to their support once they try to cheat. This isn't an overly realistic scenario, but fully in the byzantine model - the byzantine attacker will do everything to make the protocol violate a safety goal, and helping a selfish validator will achieve exactly that, even if the byzantine attacker has no additional gain. To combine those models, we thus need to either lower the number Byzantien corruptions, or assure that the attacks from the selfish validators are invisible to the byznatine ones.
- Accidential Collaboration is always possible. If the selfish validators see an easy target to frontrun, they all could individually jump on it, than thus create the same effect as a coordinated attack.How big this effect ends up depends on what the best selfish action is - if all selfish validators try to push insert their own transaction, they would just be in each others way. If they all decide to try and delay a specific transaction past the one they're inserting themselves, the effect is similar to a coordinated attack.

3 Wendy Outline

In the following, we describe the framework and the fairness protocols. As an example, we recall the block order fairness as defined in [3, 4]; this states that if a transaction tx_1 has been seen by all honest validators before another tx_2, then tx_1 must be in the same or an earlier block than tx_2.

3.1 Voting and Sequencing

When a validator sees a transaction r for the first time, it assigns a sequence number to it and multicasts that number and the transaction to all other validators. The rest of the protocol only works on complete sequences, i.e., if a sequence number is missing, all following transactions are stored, but not processed. Transaction/sequence pairs are signed, so the receiver can prove that they saw the validator sending it those transactions in that order. In addition, every transaction is accompanied by additional relevant information where available, e.g., the time of receipt.[1] Thus, every validator now has a (signed) sequence of transactions of all the other validators. Note that there is no need for agreement in these sequences – if the underlying multicast protocol allows validators to send different sequences to different validators, the protocol still works (though those validators are easily identifiable as not following the protocol).

3.2 Block collection

For every request r that is known but not scheduled, a validator computes a set $\mathcal{B}_r$ of transactions that might have to be in the same block as r. Once more information about the sequences of other validators comes in, previously unknown transactions can be added to $\mathcal{B}_r$, and older transactions can be removed. To compute these sets, it uses the *blocking functions*, which are also the functions that define what fairness means. A validator considers a transaction r blocked if, given then current knowledge of that validator, there could be a so-far-unseen transaction that by the current rule of fairness needs to be in the same (or earlier) block as r. A set $\mathcal{B}_r$ is blocked if any transaction $r' \in \mathcal{B}_r$ is blocked. The blocking function is the only part of the protocol that requires the actual definition of fairness. It is also the part where the labels come in – a transaction is never blocked by a transaction with a different label. Also, different labels can use a different definition of fairness and thus a different concept on blocking. Transactions labeled as not requiring fairness are not blocked by any transactions, and can thus skip blocking entirely, and don't even cause voting messages. In the protocol description here, we call the evaluation of the blocking function whenever a voting message arrives. It is also possible to optimise this and call it only once the blockchain is ready to consume a new block. This will save the validators some computation, though it should have a

[1]This can be optimised; it is not always necessary to send the full transactions around. Also, care needs to be taken that a missing sequence number doesn't result in a buffer overflow for nodes storing the incomplete sequence. Also, note that incompatible sequences – e.g., timestamps that don't match the sequencing – are rejected

relatively modest impact for most fairness definitions.

3.3 Delivery

Once a set $\mathcal{B}_r$ is not blocked, it is ready for scheduling. This set is then handed over as an input for the underlying blockchain layer (with sufficient proof that it has been generated properly), and all transactions in the set are deleted from all other sets. Formally, Wendy maintains two sets, the queue $\mathcal{Q}$ (which contains all the blocks that the underlying blockchain layer can pick up as well as the order in which it needs to be picked up[2], and the set of delivered transactions $\mathcal{D}$ which contains all transactions that are finished and should not be processed anymore.

3.4 The Protocol

In the protocol definition, we assume that transactions are sent to all (or some) validators by external parties, which could be collaborating with the adversary. Furthermore, we have an underlying blockchain protocol that, once it can process a new block, proposes all transactions ready for delivery that it has not delivered earlier.

$\mathcal{Q}$ is the queue of blocks that need to be scheduled by the underlying blockchain. This queue is used by the underlying blockchain, and the blockchain can delete already delivered transactions from $\mathcal{Q}$. $\mathcal{D}$ is the set of already delivered requests. This differs from $\mathcal{Q}$ in that it is used to determine if a transaction is new, so we don't process transactions twice. Furthermore, we maintain a set $\mathcal{U}$ which is the set of known and yet undelivered requests. $H(r)$ is helper data needed for some fairness functions, e.g., additional timing information.

We assume that incoming votes are stored in a data structure that is available to isBlocked and isBlockedBy. Votes that arrive out of order (i.e., with a future sequence number) are stored separately and processed once the sequence numbers fit. While this is not strictly necessary, the description of the two blocking functions is significantly simpler if they can assume all votes come in the appropriate order.

The voters are the set of parties that get messages from clients and whose incoming sequence defines fairness. The leader(s) are parties that are allowed to propose blocks to the blockchain. These two sets don't necessarily need to be the same, or even overlap. For a voting based blockchain such as Tendermint, it is sufficient if only the party that knows that it will be the next leader executes the leader part. For a proof-of-work blockchain, the voters can be a completely separate group serving as a form of oracle, while all miners execute the leader

[2]There is some freedom here again as blocks related to different labels can be processed in any order; see Section 4.2

part which requires no communication, in which case the votes should not be send as a multicast, but be requested on demand by the miners.

For the ease of description, we will talk about validators as if all validators are both voters and leaders throughout the text; also, in the protocol description, we assume all leaders went through the voting protocol, i.e., have their data structures managed according to that part.

```
Pre-Protocol Wendy for protocol instance $ID$
All voting parties:
  let $i$ be the counter of incoming requests, starting at 0.
  on receiving a transaction or vote message do
      if the message contains a request $\hat{r} \notin \mathcal{U} \cup \mathcal{D}$,
          if the request is labeled for no fairness, add it to $\mathcal{D}$ and $\mathcal{Q}$
          else send the message ("VOTE",$ID$,$b$,$i$, timestamp($\hat{r}$),$H(\hat{r})$,$\hat{r}$) to all
              parties, where $i$ is the sequence number of that request
              add $\hat{r}$ to $\mathcal{U}$
  on receiving a valid delivered block B from the underlying blockchain do
      put all elements from B into $\mathcal{D}$ and remove them from $\mathcal{U}$ and $\mathcal{Q}$
      postprocess B
All potential leaders:
  on receiving a vote message with a correct sequence number or changing $\mathcal{U}$ do
      for all  transactions $r \in \mathcal{U}$, set $\mathcal{B}_r$ to $\{r\}$
      while for any $\mathcal{B}_r \neq \emptyset$ any request $r' \notin \mathcal{B}_r$ blocks a request $r'' \in \mathcal{B}_r$
          add $r'$ to $\mathcal{B}_r$
      end while
      for all $r$ for which no request in $\mathcal{B}_r$ is blocked,
          add $\mathcal{B}_r$ to the $\mathcal{Q}$, 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 $\mathcal{B}_x$ and $\mathcal{U}$
```

There are two core functions to the protocol that define both what is considered fair and impose most of the model. For the ease of description, we assume here that validators postpone out of sequence votes, i.e., if a vote with sequence number s is accepted, all votes with sequence numbers smaller than s have been seen. This is not strictly necessary, but makes arguing about the protocol much easier.

3.5 isBlocked(tx)

The function isBlocked (tx) identifies if it is possible that a so-far-unknown transaction might be scheduled with priority to tx. If this is the case, tx cannot be consumed by the blockchain. A transaction usually is blocked if there are missing votes from other validators concerning transactions that have been seen before tx.

In the order-fairness definition, tx is blocked if it has received t or less votes;

this implies that it is still possible that $n - t$ votes come in that report to have seen a transaction before tx that currently has not been seen.

We assume that isBlocked is monotone, i.e., once a transaction is unblocked, it cannot become blocked anymore if more information comes in. This assumption should derive straight from the definition, as a function is blocked if there is only the possibility that it needs to give priority to an unknown transaction, i.e., it assumes the worst case about all data it does not have yet. While the protocol will still operate with non-monotone blocking functions, the results may not make overly much sense.

3.6 isBlockedBy(tx_1, tx_2)

This function determines if tx_2 might have priority over tx_1, i.e., if (assuming all still missing votes are worst case for tx_1, by the fairness rules tx_2 must come in an earlier or the same block as tx_1.

In the order fairness model, tx_1 is **not** blocked by tx_2 if there are $t+1$ votes reporting tx_1 before tx_2, i.e., at least one honest party saw tx_1 before tx_2.

As for isBlocked, we also assume that isBlockedBy is monotone, i.e., once the function concludes that tx_1 does not block tx_2 anymore, no additional information will change that.

3.7 Correctness

To show correctness of Wendy, we need to make some assumptions on the blocking functions. It is then still required to show for each blocking function that those assumptions are correct.

Assumption 1. *If there is a transaction that a validator has not seen yet, and it is possible that this transaction has to get priority over another transaction $tx \in \mathcal{U}$, then isBlocked(tx) = true.*

Assumption 2. *If there are two transactions $tx_1, tx_2 \in (U)$, and tx_1 needs to get priority over tx_2, then isBlockedBy(tx_1, tx_2) = true.*

Given those assumptions, the protocol Wendy assures that for all transactions, if tx_1 has priority over tx_2, then tx_1 will be in the same or an earlier block than t_2.

Proof. The proof is straightforward. Suppose tx_1 has to get priority over tx_2. While tx_1 has not been seen by the validator, tx_2 cannot be put into $\mathcal{Q}$, as all sets $\mathcal{B}_r$ it is in have a blocked element (namely tx_2). If the validator has seen both tx_1 and tx_2, by the construction of $\mathcal{B}_r$, any $\mathcal{B}_r$ that contains tx_2 also contains tx_1. Thus, if tx_2 is added to $\mathcal{Q}$, so is tx_1. □

If a transaction is put in $\mathcal{Q}$, than it must have been in a set $\mathcal{B}_r$ such that not transaction in $\mathcal{B}_r$ was blocked, i.e., no for no transaction $tx \in \mathcal{B})_r$ isBlocked(tx) is true. Thus, for all transactions in $\mathcal{B}_r$, at least $t+1$ votes have been received, alongside (by assumption) all corresponding votes with lower sequence numbers.

By the condition of the inner while loop, if for any $tx_2 \in U$ isBlockedBy(tx_1, tx_2) is true, then $tx_2 \in \mathcal{B}_r$. Thus, for all $tx_2 \notin \mathcal{B}_r$, at least $t+1$ votes reported tx_1 before tx_2.

If tx_e is not $\in \mathcal{U}$, then by the definition of $\mathcal{U}$, no vote for tx_e has arrived. As $t+1$ votes have arrived for tx_s, and thus have all corresponding votes with lower sequence numbers, at least one honest party has seen tx_s before tx_e.

If tx_e is $\in \mathcal{U}$, then by assumption 2 and the definition of the isBlockedBy function, if tx_2 added to $\mathcal{Q}$, so is tx_1 unless it already has.

For termination, we need an additional assumption:

Assumption 3. *If all messages are delivered, then for every transaction tx, eventually, isBlocked(tx) returns false.*

With this assumption, we can show the following statement:

Theorem 1. *A set $\mathcal{B}_r$ either adds elements forever, or is eventually delivered.*

This is relatively straightforward; if a set B_r does not add elements forever, at some point all elements that will ever be in $\mathcal{B}_r$ have been added. By assumption 3, all of them will eventually be unblocked, and thus the set can be delivered.

Note that is it not strictly necessary for a blocking function to satisfy assumption 3, or to guarantee that elements aren't added to $\mathcal{B}_r$ forever. Alternatively, it is possible to detect that a deadlock might have occured; for example, by looking at the age of an undelivered transaction of the size of the $mathcalB_r$, and then call the switching function to (temporarily) change to a different blocking function that can resolve the deadlock.

3.8 Block Postprocessing

While Wendy only assures block order fairness, a block contains enough information to also assure (some) fairness between individual transactions. If we extend our definition to say *if all honest parties have seen tx_1 before tx_2, then tx_1 needs to be schedule before tx_2 whenever this is decidable*, the block preprocessing function can analyse if an undecidable situation exists, and sort all remaining transactions according to the appropriate order.

One should note that this approach is not entirely trivial. As it is now, given above definition, it is possible a block of undecidable transactions will dissolve with further votes coming in. Without block postprocessing, this is sufficient for Wendy to forward all these transactions to the blockchain. Given above definition, however, Wendy needs to make sure that the block post-processing function also has all the necessary votes available to know for sure if an undecidable situation existed or not. Thus, the *isBlocked* function would need to hold back an undecidable set of transactions until enough information is collected to allow the post-processing algorithm to do its job. It is possible to keep a transaction delayed as long as possilbe (i.e., every transaction is blocked until it got $n - t$ votes) to gather the maximum amount of information, though that would add to the latency caused by the protocol.

The specific definition given above has some other interesting issues. In an asynchronous system, undecidability itself can be undecidable (as we don't know it there's still a vote coming in that makes it decidable). If the blocking function assumes some synchrony, it can work around that by blocking until all votes have either arrived or caused a timeout.

A final aspect that is worthy of future investigation is to add additional policy to the postprocessing order. There has been some work on centralised exchanges to counter the timing arms race, for example, by adding a (random) valie of $1 - 10$ milliseconds to the timestamp of each received transaction, which eliminates the businesscase to spent millions to shave of a nanosecond. As a similar armsrace is possilbe here (with cleints sitting on a highspeed connection at the minimum distance to the majorities of validators, for example), it would be very worthwhile to consider similar measures beforehand to avoid that setting from the beginning.

4 Additional Functionality

4.1 Labels

Each transaction touched by Wendy should have a label indicating a 'fairness-group'. Only transactions that are in the same group need to be fair with respect to each other – thus, we do not create unnecessary latency by sorting unrelated transactions. It is also possible that a transaction has several labels, which would then connect the two labels to some extent. For example in a distributed currency trading platform, each currency may be assigned a fairness group in order to enforce the current definition of fairness between transactions that touch the same currency, but not enforce fairness between transactions between a non-overlapping set of currencies.

The implementation of the labels is essentially done in the *blocking function* described in Section 5.

A transaction can also have several labels. In this case, it needs to be relatively fair to both blocking functions before it can be scheduled, and can also block transactions from both labels itself.

4.2 Multiple Queues

As the protocol is described now, we have one queue for all transactions passing through Wendy. This gives the underlying blockchain the illusion of dependencies that are not there – transactions from different labels are all in the same queue, and the blockchain only knows to process the queue in the right order. An alternative is to have separate queues for each label, and then let the blockchain pick transactions from each queue. This way, if a block is near full, it is easier to pick a small element that still fits in, rather than having one big set clogging up everything.

This approach opens an interesting point in inter-label fairness. Once each label has its own queue, we can also give some queue preference over others, e.g., prefer a high-paying trading market over a causal game. It also would make it easier for an application to support 'unimportant transactions' that will be processed if the blockchain has sufficient capacity, but can be dropped if there is insufficient capacity.

Multiple queues are somewhat in conflict with supporting transactions with several labels; in this scenario, either those labels need to share the same queue, or some transactions need to be put into the queue belonging to the other label to maintain fairness.

4.3 Protocol Switch

There are a number of situations where a protocol might want to change the definition of fairness on the fly. This could be because some definitions of fairness can't guarantee a maximum block size, and thus need to switch to a weaker definition to assure fast termination. In addition, external conditions might change, e.g., a market crash that requires special attention. The protocol allows for an easy switch – essentially, all that needs to happen is to take all unscheduled known requests r and recalculate their corresponding $\mathcal{B}_r$ using the new fairness definition.

4.4 Validity

Any block has to come with a proof of validity; this proof is essentially the signed sequences that lead to the creation of that block. To optimise computing time, a validator can remember signatures it already verified; it should have received most parts of the sequence earlier in the protocol. The validity verification is one of the points where the protocol interfaces with the underlying blockchain – a block in the blockchain is only valid if all subblocks generated by Wendy pass their validity functions. The block validity function also needs access to the blocking function to be able to evaluate if it has been applied properly, and needs to know if (and with what justification) a protocol switch was performed.

5 The Order Fairness Evaluation

In the context of Wendy, we first need to provide block level fairness, i.e., if we'd need to schedule r before r', Wendy assures that r is in the same or an earlier block than r'. Detailed scheduling of requests inside the block can then be done by the post-processing of the block based on the information associated with the corresponding transactions.

Wendy provides a framework protocol that can enforce different fairness definitions, separate streams of transactions that need to be relatively fair to each other (identified by labels), use different definitions of fairness for different labels, and switch the definition of fairness that is used for a particular label on the fly.

The definition of order fairness comes in through the blocking function, which determines if a transaction is blocked by another one (i.e., cannot be scheduled unless the other one is scheduled in the same or an earlier block), or if a transaction might be blocked by a yet unseen one. While the Wendy framework does not have a lot of assumptions on the validators and network, the blocking function might introduce stricter requirements for their definitions of fairness, such as a limit on the number of corruptions, the existence of trusted time, or some level of network synchrony. The blocking function also implements the enforcement of the labels – a transaction can only be blocked by a transaction that has the same label. Also, the label indicates which fairness definition is used by the blocking function.

Wendy will always ensure order fairness as long as the assumptions underlying the fairness definition hold. However, for some definitions, it is not possible to always ensure liveness – for some definitions of order-fairness, we cannot guarantee that the protocol terminates. To mitigate this, we introduce conditional unfairness. This allows us to define a condition (e.g., too many transactions in one block) under which fairness can be ignored for one block to resolve the deadlock and then switch back to the original definition. Alternatively, it is always possible to temporarily switch to a weaker definition of fairness.[3]

We assume that there is a known set of validators. A client sends a transaction to one or several validators, which then multicast it to its peers that might not have seen the transaction. While definitions of fairness can vary, we usually define a fair order through the order or the timestamp at which all honest validators have received the transaction, even if we do not know what validators are honest – for example, one straightforward (and unfortunatelly too strong) definition is that if all honest validators see r_1 before r_2, then r_1 must be scheduled before r_2.

[3]We can also make the condition randomised to make it harder for an attacker to control where the unfairness hits.

5.1 Block Order Fairness

We now describe some fairness definitions that Wendy supports. While Wendy can always ensure that order fairness is satisfied, some definitions of fairness need extra requirements to either provide a meaningful concept or to guarantee termination. The most straightforward definition for fairness is that *if all honest parties saw tx_1 before tx_2, then tx_1 needs to be scheduled earlier than tx_2*. Unfortunately, is has been shown in [3] that this definition allows for a set of transactions to be undecidable (essentially, the correct order depends on who is dishonest, which is not known to honest parties), and in [4] that these undecidable sets can have an unlimited size. Thus, the definition used for Wendy is as follows:

Definition 1. If all honest parties see transaction r before transaction r', then r and r' need to be in the same block.

Let t be the maximum amount of traitors the protocol tolerates. A request r blocks abother request r' with the same label if it is possible that $n-t$ parties saw r before r', i.e., if $n-t$ parties either reported r before r' or have not reported r so far.

A request r is blocked if it is still possible that a previously unknown request has been seen by $n-t$ parties before, i.e., if $n-t$ parties have not reported r yet.

This definition can tolerate corruptions up to $n/2-1$; in a synchronous system, it can even tolerate an arbitrary number of failures, though a larger t will lead to larger numbers of transactions that need to go into the same block and make it easier to create an endless schedule (see below).

5.1.1 Termination

We can show that in the strictly asynchronous model, relative order fairness cannot guarantee termination even if only one party can be corrupt. An adversary with a very high degree of network control is able to create transactions in an order that an unlimited number of transactions end up in the same block. This can be mitigated either by using the protocol-switch function – if the block gets too large, a weaker definition of fairness is (temporarily) used to resolve the situation, or with a slightly weaker timing model that restricts the adversary's ability to completely determine the order in which messages arrive.

5.2 Timed Order Fairness

Definition 2. If there is a time τ such that all honest parties saw r before t and r' after τ, then r needs to be in the same or an earlier block than r'.

Let t be the maximum amount of traitors the protocol tolerates. A request r blocks another request r' if it is possible that $n - t$ parties timestamped r before some time τ and r' after τ, i.e., if such $n - t$ parties either issues such a timestamp, or have not reported any request with a timestamp higher than τ.

A request r is blocked if it still can be blocked by an unreported request. This is always the case if r has been reported by less than $t + 1$ parties. If r has been reported by $t + 1$ or more parties, then let τ' be the $t + 1$ largest largest timestamp of r. Now r is blocked by a request r' if r' has $n - t$ validators either reporting r' with a lower timestamp than τ or have not reported r', and the last request they reported has a lower timestamp than τ'.

Note that while this requires local clocks, there is no strict requirement on clock synchronisation – the definition becomes more meaningful the more closely the clocks are synchronised, but the protocol will still operate with completely unsynchronised ones. In fact, the only two requirements for clocks that we do have is that a clock always counts forward and that it does so by a minimal amount (thus, the time cannot converge towards a finite number). It is also possible to go completely clock-less, i.e., to use the transaction counter as a clock.

5.2.1 Termination

Termination is assured as every transaction becomes unblocked as long as new transactions come in so new timestamps are generated; similarly, for each transaction tx, eventually new messages will have a timestamp where they can't block tx anymore.

5.3 Capitalistic Fairness

For this definition of fairness, every transaction includes a transaction fee the corresponding client is willing to pay; this is in some sense similar to the Ethereum model. For the purpose of this description, we do not worry about the format and the enforceability of that transaction fee, but simply assume that every transaction has a corresponding number. For this definition of fairness, the blocking functions need to synchronise with the blockchain; either, the evaluation should only be invoked once a new block is ready, or we need some information about the state of the chain; for this definition, we use the set $\mathcal{Q}$ to synchronise.

A straightforward definition for the blocking function would be as follows: If $\mathcal{Q}$ is non-empty, everything is blocked, i.e, a leader does not process transactions until the underlying blockchain is ready to absorb them. If $\mathcal{Q}$ is empty, then every transaction is blocked by any transaction that has proposed a higher fee.

There are a couple of issues with this definition. For one, we might end up scheduling only one transaction per block, which is highly inefficient. This is one reason for the structure proposed in Sectionrefsec:mqueues. Wendy keeps giving queues to the underlying blockchain until the block is full.

he way we defined the blocking function above does allow validators to cheat to some extent though – a validator can pretend to never have seen a high paying transaction and schedule a low paying one nevertheless. To avoid that, we can modify the second part of the blocking function as follows:

If $\mathcal{Q}$ is empty, then every transaction is blocked by any transaction that has proposed a higher fee and that has been voted for by at least $t + 1$ parties.

5.4 Capitalistic Fairness with Social Security

The capitalistic fairness definition has the problem that low paying transactions might never get executed. To give paying customers an edge, we can add a waiting factor. To this end, the blocking function would maintain a counter for each transaction it is aware of that has not been scheduled yet. Every time a transaction is added to $\mathcal{Q}$, all remaining transactions increase their waiting counter. The fee offered with a transaction is then weighted taking this counter into account, e.g., multiplied with $\frac{1+counter}{100}$ as a linear weight, or $1.01^{counter+1}$ for an exponential increase in weight. Another model would be to weight relative to the medium received time of the voting parties, and thus increase weight as absolute time passes.

5.5 Lottery

The Lottery model is a variation of capitalistic fairness which chooses a random subset of the seen transactions, where the probability (can be) weighted by paid fees.

If $\mathcal{Q}$ is non-empty, everything is blocked, i.e., a leader does not process transactions until the underlying blockchain is ready to absorb them. If $\mathcal{Q}$ is empty, then a pseudorandom function is used to establish an order of all known transactions, and every transaction is blocked by all transactions that come before it in the order.

5.6 Conditional Order Unfairness

Conditional unfairness is essentially not blocking any request if a predefined condition is satisfied, e.g., a request being in a block for that label exceeding a maximum size, or the last time a request for that label being scheduled exceeded a specified timeout (all blocks not satisfying the condition are blocked by default). Conditional unfairness is only used to resolve a deadlock by scheduling the offending block, and then should revert back to the original fairness definition. The condition can also determine which transactions are scheduled

– if, for example, the condition is that a block must not have more than 100 transactions, then all transactions in blocks of that size would be scheduled next.

5.7 Purpose Bound Fairness

A final interesting form of fainress is purpose bound fairness. If we consider, for example, a market for derivatives on Australian beef prices, the primary goal would be to allow actual stakeholders in Australian beef to use it as an insurance against price fluctuations. A different use would be high-speed traders with now actual interest in beef to speculate on price fluctuations and thus get a nice cut. Once could consider a fairness rule that prefers the former over the latter – while high speed trading is still fully possible, a party with an actual stake in the market would always get preference, and thus not be drowned by traders with no stake but fast networks.

This also raises the possibility of interesting follow on work on how to prove that one is an Australian cattle farmer on the internet, which we leave for future work.

5.8 Tier Model

The tier model is an add-on to all other models that introduces different preference tiers. Suppose we have two tiers (priority and normal). In addition to the normal blocking, Wendy would add one additional constraint, namely that no transaction in the priority tier is ever blocked by any transaction in the normal tier.

6 Additional Properties and Features

6.1 Multi Labels

A transaction can have multiple labels and thus be required to be order-fair with respect to all of them. This would however mean that those two labels are linked – the blocking function now needs to also verify if any transaction on $\mathcal{B}$ is blocked for any of its other labels, and recursively verify the blocks of the other IDs.

The point where this gets complicated is the protocol switch. If one market is designed for speed and rather sacrifices some fairness to improve performance, but one transaction in it is part of a very conservative market that doesn't, then the slow protocol will dominate the fast one.

6.2 Blocksize & Communication with the Blockchain

As Wendy is meant to be chain agnostic, we cannot make an assumption on the blocksize of the underlying blockchain. Ideally, the blocksize of that chain is significantly larger than the maxium blocksize produced by Wendy, in which case the blocks generated by Wendy should easily fit into the blockchain. A more complex alternative are virtual blocks, i.e., a Wendy-block could be scheduled in several blocks of the underlying blockchain and reassembled later into a bigger block. There are a number of additional issues in the communication with the blockchain that still need to be sorted out. If the blockchain has enough capacity to always consume the entire queue, it does not need to worry about order. If it cannot do so, however, it is vital that the blockchain maintains the structure of $\mathcal{Q}$, i.e., processed the entries of $\mathcal{Q}$ in the order in which they where entered, and does not cut off in the middle of a set. While it is not catastrophic if that doesn't happen – it is always possible to re-sort things after the block has been delivered, as the validating information contains enough information – this would cause extra effort and latency. Unless the underlying blockchain supports some form of grouping and priorities, the easiest way would be if Wendy is kept in control. This would mean that the blockchain reports the amount of space it has left in the new block, and Wendy then feeds as much of $\mathcal{Q}$ into the mempool as fits into that space.

6.3 Message Loss

As the protocol is defined now, it assumes that no messages from voters are lost. A lost message will result in a bad sequence number, and thus block that voter for good. On the other side, this structure also makes it easier to mitigate message loss – the recipient of a vote knows if an message is missing (or delayed), and thus can react on this. There's a number of measures an implementation can do to make the protocol more robust. Most obviously, some resend mechanism can be built in to detect a missing vote (which is comparatively easy) and then ask the corresponding validator to resend it. An additional (and faster) mechanism would be for every voting message to include the hashes of the last, say, ten votes in sequence. A validator that got the full transaction otherwise (through other votes or directly from a client) can thus easily compensate for missing votes, while the communication overhead is comparatively modest.

7 Performance Impact

The Wendy protocol is executed in parallel to the hosting blockchain. Thus – apart from sharing bandwidth and computation power with the validators – there is no added latency for the blockchain protocol itself. The main impact is that transactions that are affected by the fairness protocol may be pushed to a later block than they would normally appear in, as Wendy needs some time to process and clear them. For transactions that are marked to need no fairness, this delay does not apply; Wendy passes them right into the blockchain's mempool, thus causing practically no delay (we assume here that the execution time of the code required to check the message label and forward it to the mempool is negligible). For all other messages, there are three sources of delay, where our numbers where taken from order-fairness:

Voting Time. By most sensible fairness definitions, a transaction needs to receive one or several votes before it can be unblocked. This adds one round of communication to the processing time of a transaction. However, this does not necessarily delay the transaction in the end, as the underlying blockchain also needs time to finish a block and get ready to empty the mempool, which gives most transactions time to finalise the voting. In our simulations, the ratio of transactions that ended up delayed through this effect was very close to the ratio between (average) message transmission time and the block generation time of the underlying blockchain, which is exactly what one can expect here. Thus, Wendy has a big effect for fast blockchains, and a very little one on slow ones – if Wendy is used with Ethereum, the effect should be barely notable, while blockchains like Tendermint might delay 5-10 percent of the transactions.

Blockings. To assure fairness, some transactions have to be blocked by others. This in itself does not create any delays yet, as Wendy would just ensure both of these transactions end up in the next block together. The only point where this causes a delay is if the blocking transaction is also blocked due to insufficient votes (i.e., delayed through the voting time). While theoretically this effect can cause an unlimited delay, our simulations showed that this effect delays between 15 to 20 percent additionally to the transactions blocked by the above effect.

Blocked Votes. A factor of a different kind are out of sequence votes. As votes need to be processed in the order they are sent (by sequence number), a voting message that for some reason is massively delayed thus holds back some of the succeeding vote messages. The effect is that for computing the delay through voting time, what matters is not the average message delivery time between two parties, but also the worst case.

7.1 Reducing the message complexity to $O(n)$

One optimisation that will drastically improve the communication complexity is to only send voters to upcomming leaders. This work especially well on relatively slow blockchains, where the number of leader that have a reasonable probability of getting involved in proposing a particular transactions is very low. Thus, the message complexity would go down from n^2 to $O(n)$, with a constant around $2-3$. To make this work, two new features need to be implemented. Firstly, if a transaction is blocked longer than expected and thus is processed by a leader that wasn't initially expected to process it, votes need ot be resend. This can be done relatively easily though, as voters can see in the blockchain that their transaction has not been scheduled yet, and thus deliver the corresponding votes; in fact, it is possible to always send votes only to the next two leaders (assuming a protocol where they are known), and only send them to a following leader if they did not appear in the next block. The second feature that needs to be implemented is to allow future leader to keep their sequence numbers updated. If a leader misses a voting message, in the current implementation they would have a gap in the sequence they saw, and thus would not be able to process new votes. This, too, can be fixed by watching the blockchain. As every fair transaction is validated by the signed sequence numbers of the corresponding voters, those sequence numbers are part of the blockchain. A validator that did not see the vote directly thus does see the sequence number through the blockchain, and – knowing that it does not need to process the corresponding transaction anymore – can thus consider that sequence number as processed.

With these steps, the message complexity would be $2n$ in the normal case (rather than n^2, with some added multicasts in case of transactions being delayed. One should be careful about a side-effect for the clients though. In the original model, it was sufficient (though fairness- latency-wise unwise) for a client to send a transaction to only one validator, as the validator would multicast it right away (assuming it is honest, which should be the case most of the time). In the new scenario, the transaction would still slowly propagate between the validators, but require almost n blocks to reach all validators. This is assuming that validators and leader are one of the same; otherwise, the client would need to multicast his message anyhow (or the voting validators need ot multicast among each other as well as to the leader validators).

7.2 Measuring Parameters

Our measurements where made on a simulator that has most of the Wendy implementation (without signatures and validation, and with all parties behaving honestly) on top of a simulated network and blockchain. The network would place validators and customers on a random location on a flat (and square) earth, and use the distance between the parties as basic message delivery time. In addition, a parameterizable random component was added to all delivery times, with the exception of parties sending messages to themselves. The blockchain has its own timer, and would deliver its block, pick up transactions for the next block, and restart the timer. We assumed that the blockchain can consume all transactions it needs to, i.e., no transactions are left in the mempool. For the simulation runs we chose a blockchain that only starts processing new transactions once the last block is finished, as this is the most common model today. We can also simulate parallelised blockchains that start a new block before finishing the previous one; as this would result in the blockchain picking up transactions faster, it should increase the number of transactions delayed through Wendy.

7.3 Impact on Longest Chain Protocols

Wendy has a voting based approach, and thus does require voters to send messages to all potential block leaders. In a protocol where that number is rather large, this can be an inhibiting factor. There's a number of ways this can be dealt with.

- On a general purpose chain (such as Ethereum), we can expect that the vast majority of transactions do not require fairness; fairness would only be needed by specific applications. As non-fair transactions don't need to touch the pre-protocol at all, the added communication load would only apply to a small fraction of transactions.
- If we separate voters from leaders, rather than the voters sending their votes to all parties, they can send them to each other, and then send the combined votes to the leaders (or offer them for a pull request). This saves some bandwidth (as the transaction content only needs to be sent once), and saves a lot of message overhead. A similar model would be for the client to (also) send a transaction to the leaders, which then pull the fairness votes from the voters once available; this way the voters only need to send the appropriate signatures.
- Wendy could run completely separately; transactions in $\mathcal{Q}$ could then be pulled by the blockchain miners.

As Wendy was originally designed to be integrated in high speed voting based protocols, the interaction with longest chain based protocols still needs work; the next steps are now to engage with the corresponding communities and figure out what the best way is to adapt Wendy to their requirements.

8 Fairness vs. Causal Order

A orthogonal approach that can be taken to protect against frontrunning is causal order. In this approach, frontrunning is still allowed, but it is only possible to do based on the existance of a transaction, not based on it's content - the content is only revealed once the transaction has been scheduled and thus the order is fixed.

A client can add causality protection to a transaction. This protection uses a different mechanism than order fairness; essentially, it encrypts the transaction until it is scheduled. This can be applied to any transaction, independent of the context it is made in. The encryption is a threshold encryption scheme, i.e., envrypting one plaintext generates n cyphertexts, t+1 (or n-t) of which are required for decryption.

If a transaction is marked as causality protected, the client threshold-encrypts the message, and sends the shares as long with a unique identifier (e.g., the hash of the transaction) to the individual validators. The validators only process the hash, while the actual transaction is only revealed later.

8.1 Causal Order without Fairness

The issue with this scheme is that we need to assure that the transaction can be decrypted before it can be scheduled; this is difficult to synchronize properly without risking delaying the entire block to find out what the transaction is. On the other side, the decryption cannot be released too early - the blockchain can have issues right until a block is scheduled and end up not scheduling the corresponding block. Thus, we propose to allow a client to choose between different levels of protection:

Light protection: The decryption shares are broadcast as soon as the leader has send the first proposal for the block. This guarantees that they are available by the time the block is finished, but risks early exposure - if the blockchain times out and replaces the leader, the content of the transaction is known and it can be frontrun. Also, this only works reliably if a t+1 threshold is chosen. If n-t shares are needed to sucessfully decrypt, it might be possible that some validators have the ability to decrypt, while others do not.

Strong protection: The hash of the message is scheduled in the blockchain. Once it is in a block, the validators publish their decryption shares. The transaction is then considered delivered in the following block, i.e., it looses one block in latency. An open question is how to combine this with order fairness - if we want to be strict, all trandactions that need to be order-fair with respect to this message also need to be delayed by one block (we can make this optional to the market maker to allow this, or decide that if a transaction has strong causality protection, it will be treated slightly unfair (i.e., be one block late) to prevent it from delaying everyone else.

There also is the option that a client not only hides the content of a transaction, but also its own identity. This can be necessary in settings where a tradrs past can carry information about a likely future intent, or where the fact that a specific account is active in itself is an issue – if, for example, a Bitcoin adress linked to Nakamoto would do a transaction, the pure fact that this adress would be used would be sufficient information to affect prices. There are some sublte issues if we cannot identify a client. For example, allowing such transactions with no constraints would make it extremely easy to spam the network, and fill it with bogus blocks. It would therefore make sense to require a deposit for this kind of transactions, and keep it if it does not decrypt properly or doesn't meet certain quality conditions. One also could in general charge a comparatively high fee for hidden identities, as it is safe to assume that they only make sense for a high volume transaction, and anyone prepared to make that transaction should be willing to pay the price (we do leave out at this point the issue of anonymous payment of the fee; this can be done through some zero knowledge techniques, or the client simply has to create a second, anonymous account and use that one for payment; as serious anonymous users do need to require extra work anyhow (e.g., mask their IP adress), this is not an outrageous requirement).

8.2 Causal Order in Wendy

If a transaction is also subject to the fairness rules of Wendy, causal order becomes much easier to implement, and can be integrated with much less overhead. On a normal blockchain, the content of a transacion can only be revealed once it is in a finalized block, adds latency and complexity. If we use a fairness protocol, the content of a transaction can already be revealed once it is not possible (or at least unlikely) that another transaction can frontrun it. If we assume block-order fairness, we can see two implementations of causal order:

- The client can send a $n - t$ threshold encrypted transaction to all validators, which immediatelly reveal their encryption shares together with their vote. While this does not assure causality, it makes a practical frontrunning attack quite hard; assuming we have the optimized version of Wendy where votes are sent to only one leader, that (corrupt) leader needs to wait for at least $t + 1$ honest votes to decrypt the payload, then send their own transactions to the remaining t honest parties in a way that it arrives before the message of the original client; if the leader does not collaborate with the full number of allowed corruptions, this is getting even harder. While in our theoretical model this attack is completely feasible, its practical hurdles might be enough for many applications, given the advantage of no added latency or communication overhead.
- If we use the slightly more conservative version of the protocl where a transaction tx is unblocked once it got $n - t$ votes, the validators can open their shares once they see a valid proposal for a block containing that transaction. As $n - t$ parties have already committed to a sequence containing tx, the only way a new transaction can frontrun tx now is if some corrupt validators sign incompatible sequences. In this setting, hoever, this would immediatelly become aparrent, and any block that contaons the frontrunning transaction would be rejected by all honest validators who have seen the original already.

For the timing based protocol, a similar approach can be taken. Seeing the timing information about a transaction in the votes, any validator can compute a safe time after which a frontrun is no longer possible, and publish theirdecription shares once that time has passed.

References

[1] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., and Juels, A. Flash boys 2.0: Frontrunning, transac-

tion reordering, and consensus instability in decentralized exchanges. *CoRR abs/1904.05234* (2019).

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

[3] Kelkar, M., Zhang, F., Goldfeder, S., and Juels, A. Order-fairness for byzantine consensus. Cryptology ePrint Archive, Report 2020/269, 2020. `https://eprint.iacr.org/2020/269`.

[4] Kursawe, K. Wendy, the good little fairness widget. *IACR Cryptol. ePrint Arch. 2020* (2020), 885.