Heterogenizing DAG-based consensus ($1/n$): heterogeneous narwhal-rider

Typhon Team Heliax

March 30, 2023

Abstract

Blockchains exhibit *linear* structure as each block has a unique reference to *the* previous block. If instead, each block may reference *several* previous blocks, we can build a *directed acyclic graph* (DAG) of blocks.[1] In fact, such "block DAGs" are the basic data structure that several recent consensus protocols rely on, *e.g.*, DAG-rider, Bullshark, and Narwhal&Tusk.[2] These protocols build a growing "global" DAG of transaction data such that—among other things—(1) every validator's local view is a sub-DAG of an "ideal" global DAG and (2) every block of the "ideal" DAG is *logged* for inclusion into a total order of blocks; the latter typically implies (eventual) execution of the respective transactions.

The paper focuses on the general principle of heterogenization and presents the general idea by means of a specific example, namely *Heterogeneous Narhwhal-Rider*, or HNR for short. Very much like Heterogeneous Paxos is a heterogeneous version of Lamport's Paxos, HNR is a heterogeneous version of the Narwhal mempool, extended with DAG-rider's weak links. We conclude with a discussion of how heterogenization is a suitable tool for building towards an eco-system, very much in the spirit of Vitalik Buterin's `cross-chain world`.

Contents

[1]Note that this includes block chains as one particular case.

[2]The respective references are,

1 Introduction

Directed acyclic graphs feature in several recently proposed consensus protocols [?]. So, are we presenting yet another DAG-based consensus protocol?

Clearly, the answer is no; the research is active, and somebody might be in the very process of writing a new one. The present paper rather explains how to heterogenize one example of DAG-based mempools; we believe that this process applies to the whole family of DAG-based mempools, which we shall dub DAG-pools. So, for the sake of specificity, we use a mix of Narwhal and DAG-rider as a first guinea pig, experimenting with the general principle of heterogenization.

Along the way, we shall find out that heterogenization applies differently to questions of availability and integrity [?]:

availability all relevant transaction data is promised to be available

integrity the protocol is safe, *e.g.*, no double spends, equivocation, *etc.*.

2 Context

We want to work towards a multi-chain world in which

- everyone can interact with all chains of the ecosystems, and
- there is a unique definitive state of every base ledger .

One good example of learners are *executor nodes* [?], which are in charge of updating the state of one or several base ledger .

For technical considerations, the main points (for validators and/or executors) are

- they can participate in the production of as many base ledgers as they want
- as long as they keep a record of relevant transaction data or the ensuing state changes.

The latter point roughly corresponds to the availability protocol while the former is mainly features in the integrity layer.

3 Preliminaries

We start with a short review of the central concepts and definitions that we rely on in our description of the HNR protocol. Moreover, we also discuss the preliminaries for HNR's salient properties.

3.1 Quorums: learner-specific, global, and universal

The HNR protocol is designed to take into account learner-specific assumptions, first and foremost about liveness of sets of validators.[3] We start by fixing a finite set of *learners* L; each learner holds certain beliefs and these beliefs impose restrictions on the behavior of validators that is deemed (im-)possible. Each learner is asked to commit to a set of *quorums* such that at least one of the quorums will be live at all times. In particular, the protocol description takes as input a function from learners to sets of (learner-specific) quorums. Let us mention once more, as protocol designers, we assume each learner to have commited to a set of quorums, such that one of these quorums is live at all times (and that this is compatible with the learner's believes).[4]

Definition 1 (Learner-specific quorum system)**.** A *learner-specific quorum system* is a function from learners to sets of quorums. For a learner-specific quorum system Q and a learner a, we denote the learner's set of qourums by Q_a, whose elements are the *learner-specific quorums* of a. We use q_a, q'_a, *etc.* to range over learner-specific quorums of learner a.

One might even go as far and ask learners to choose a maximal set of quorums that is compatible with their beliefs; however, for the operational aspects of the protocol, all we need is a map from learners to sets of quorums.

Now, based on quorums systems, we can formulate the remaining two core definitions. Global weak quorums are important for matters of availability: if a global weak quorum is holding a copy of a particular transaction request, then one copy should be available (as a "shared belief" of all learners).

Definition 2 (Global weak quorum)**.** A *global weak quorum* is a set of validators X that is a weak quorum for each learner, i.e., for every learner $a \in L$ and all learner-specific quorums $q_a \in Q_a$, we have a non-empty intersection $X \cap q_a \neq \varnothing$.

[3]The terminology about trust assumptions was introduced already for Heterogeneous Paxos [SWvRM21].

[4]This is formalized as `Trust Live` in the TLA$^+$-specification.

Definition 3 (Universal Quorum)**.** A *universal quorum* is a set of quorums that contains at least one learner-specific quorum for each learner.

3.2 Discussion of desirable properties

4 Overview: mem-DAGs, heterogeneity, etc.

5 Architecture and communication patterns

Hnr incorporates Narwhal's [DKSS22] scale out architecture: each validator has a unique *primary* and a number of *workers* (see Fig. 1).

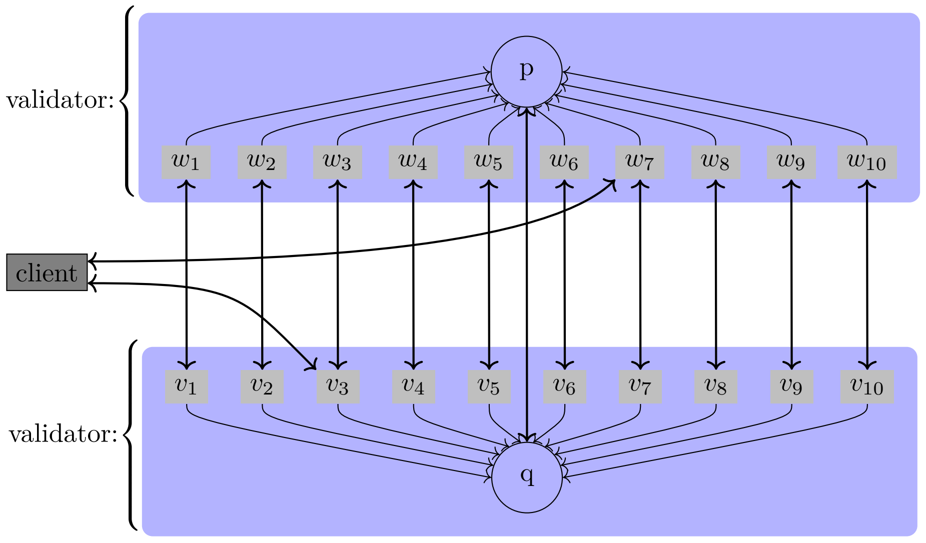

Figure 1: The structure and communication patterns of validators

6 Worker actions (availability and executability)

Every validator has the same number of workers. Thus, each worker can be assigned a unique *mirror worker* on every other validator. We shall adopt the convention that identifiers of mirror workers share the same subscript. For example, in Fig. 1, workers w_2 and v_2 are mirror workers of each other. Workers are mainly featuring in the availability protocol, which keeps the transaction available until execution. Specifically, workers keep track of (batches of) transactions, their hashes, and erasure coding shares—which shall be trivial, along the lines of Narwhal and Tusk [DKSS22]. The core information that workers provide to their primaries are hash references to transaction request data, besides consecutive batch numbers and some signatures. In this way, primaries can save network bandwidth usage. Moreover, as a secondary principle, primaries do not send messages or other direct signals to their workers.[5]

6.1 The pure availability protocol (pre-execution)

`MessageEnum::TxReq` **Transaction request collection (TX$\leftarrow$)** Each worker keeps listening for incoming transaction requests from clients.[6] Transaction requests should be buffered using reasonably fast memory (to ensure that all requests are eventually served); transaction request fees may be imposed to control the rate of client requests. worker $\leftarrow$ client

`???` **Transaction batching (TXs := TXs : TX)** Every worker stores the received transaction request to *the* current batch TXs, which is list of transaction requests. This happens unconditionally, *i.e.*, there is always a unique current transaction batch and every transaction request has to be added to the current batch. New batches may be created over time, but each batch in this "stream" of batches has a unique *batch number*. Within a batch, transactions are assigned consecutive sequence numbers: the *sequence number* of a transaction is the position in the current batch. Thus, each transaction request by identified by its *batch*, *sequence number* and the worker ID; we shall refer to this as the *fingerprint* of the transaction request.[7] [worker]

`TxToAll` **Transaction broadcasting (TX!$\boxed{\text{TX}}\Rightarrow$)** If a worker receives a transaction request, it will broadcast a copy to all mirror workers. In principle, we could use arbitrary erasure coding schemes. However, in line with the original version of Narwhal [DKSS22], we use full copies as erasure codes. Despite the coincidence of erasure codes and the "original" data, we visually distinguish the "copy" of a transaction from the "original" transaction supplied by the client, using two different symbols, namely TX and $\boxed{\text{TX}}$, respectively. When broadcasting copies of received transaction requests, the message also includes the fingerprint of the transaction request. worker $\Rightarrow$worker

`WHxToAll,WorkerHx` **Worker hash broadcast (TX!WH$\uparrow\Rightarrow$)** When a worker receives a transaction request, this might trigger a new worker hash to be produced. In principle, the decision for when to send it is for the worker to decide (as long as each worker hash contains at least one transaction worker $\Rightarrow$ worker $\rightarrow$ primary

[5]However, workers receive signals upon successful execution of transactions, which allows them to free up the transaction storage by executor nodes.

[6]The bandwith and amount of storage for storing incoming transactions *should* be big enough to process all incoming transactions. We share this assumption with Byzantine set consensus [CNG21]. Transaction fees are one way to avoid flooding attacks, making the latter prohibitively expensive. For example, we might consider using a FIFO-buffer; however a priority queue that takes into account a combination of fees and quality of service considerations is more suitable for managing the flow of incoming transaction requests.

[7]The order of transactions within a batch might be permuted before execution to avoid potential MEV.

and does not exceed a potential maximum size).[8]

WorkerHashData,WorkerHashSignature The new worker hash consists of

- the hash of the current batch TXs,
- the length of the current batch,
- the identifier of the current worker,
- a signature of the above data by the current worker.

The worker hash broadcast involves the following steps:

1. broadcasting the new worker hash to mirror workers;
2. sending the new worker hash to the worker's primary;
3. resetting the current transaction request buffer to the empty list;
4. incrementing the batch number;
5. last, but not least, storing the transactions of the current batch for retrieval.

MessageEnum::TxAck **Transaction request acknowledgment** Optionally, one may acknowledge the client's requests.

MessageEnum::TxToAll **Transaction copy ($\boxed{\text{TX}}\leftarrow$)** Upon receiving a transaction copy, the worker has to store the copy locally such that its hash can be retrieved quickly via its fingerprint, *i.e.*, worker ←worker

- the ID of the collecting worker,
- the batch number of the batch to which it belongs, and
- the sequence number within the batch to which it belongs.

Note that copies of transactions are *not* signed by the worker. Signatures are deferred to worker hash broadcast. Storing transaction copies as above will be useful for handling "foreign" worker hashes, *i.e.*, worker hashes that refer to transactions that are collected at other workers.

WHxFwd **Worker hash forwarding ($\boxed{\text{TX}}$!$\boxed{\textbf{WH}}\uparrow$)** In case, the transaction was the last missing one to match a previously received worker hash, the worker hash is send to its primary. The (pre-)condition is that all transactions that are referenced in the worker hash have already been received and stored.[9] worker →primary

WHxToAll **Worker hash reception ($\boxed{\textbf{WH}}\leftarrow$)** When a worker receives a worker hash from a mirror worker there are two cases: either it has already has received all the referenced transaction copies (and can forward the worker hash to its primary) or it has to store the worker hash (and the worker hash forwarding will be triggered by the last transaction copy). worker ← worker

[8] At which exact moment worker hashes are compiled can depend on several factors, *e.g.*, on a maximum number of transaction requests per worker hash or a maximum delay between the first and the last transaction request within a worker hash.

[9]This might be unexpected order of events, but the sending of transaction data might be "delayed".

MessageEnum::WHxFwd **Worker hash forwarding (WH!WH↑)** If a worker has already stored the transactions of a received worker hash, it sends the worker hash to its primary.[10] worker → primary

We have described the protocol in terms of what workers do in reaction to *receiving* messages. This emphasizes that sending a message might be due to several, slightly different types of scenarios (depending of which message from a *set* of triggering messages arrives last and thus becomes the final trigger).

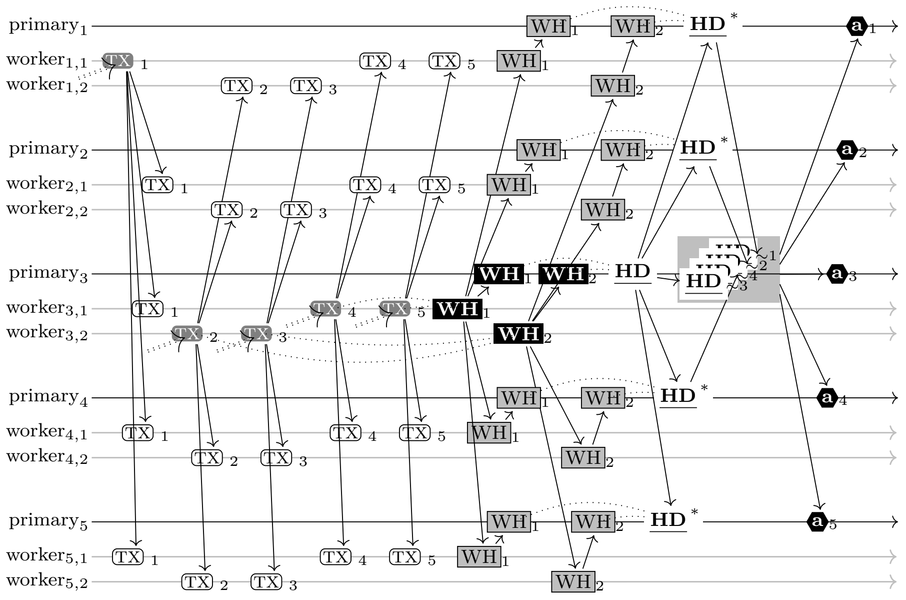

Figure 2: The availability protocol in the genesis round

6.2 Execution support protocol

7 Primary actions

Primaries will follow a protocol in which we can distinguish between matters of availability and matters of integrity. For the availability protocol, we shall treat first the special case at genesis and later describe the additional (re-)actions in the typical mode of operation. The integrity protocol is described in between the two; after all, the two protocols are closely intertwined.[11]

[10]Validators will use this information to send availability commitments to block headers of other primaries.

[11]Specifically, the integrity protocol produces blocks, and signed quorums of these are necessary for block headers (after genesis); however, header announcement and signing are part of the availability protocol.

In the protocol description for validators, we avoid copying text if a message has several triggers by simply listing the alternative triggers of a message (after the "default" trigger). For the pure availability protocol for workers, this would lead to a single heading for worker hash forwarding.

Worker hash forwarding ($(\boxed{\mathbf{WH}}/\boxed{\mathrm{TX}})!\mathbf{WH}\uparrow$)

So, the primary trigger for worker hash forwarding are worker hashes, but transaction copies are alternative triggers (in cases of message delay irregularities).

7.1 Availability at genesis

WorkerHx **Worker hash completion** ($\mathbf{WH}\leftarrow$) When a worker hash (that was compiled by a local worker) is received, the primary adds the worker hash to the current list of worker hashes **WH**s. primary ←worker

NextHeader **Header announcement** ($\mathbf{WH}!\,\underline{\mathbf{HD}}\Rightarrow$) The primary may announce the next header (if the primary considers it is "time" to do so—cf. Footnote 8). The genesis header consists of the current list of worker hashes, tagged with the identifier of the primary; the round number zero is implicit. The message of the batch announcement only contains the *fingerprint* of a header, namely primary ⇒primary

- the identifier of the primary
- a list of pairs of a batch number and a worker ID.

The actual header itself consists of

- the identifier of the primary and
- the list of worker hashes.

WorkerHx **Worker hash reception** ($\boxed{\mathbf{WH}}\leftarrow$) When a worker hash that stems from a worker on a different validator is received, the primary adds the worker hash to the current list of known worker hashes. primary ←worker

Header signature commitment ($\boxed{\mathbf{WH}}!\boxed{\boxed{\underline{\mathbf{HD}}}}_{\sim}\rightarrow$) If the received worker hash completes the list of known worker hashes to match a previously received header fingerprint, the primary commits to the header by signing the header and sending the signed header back to the creator of the header. primary →primary

Header announcement ($\underline{\mathbf{HD}}^{*}\leftarrow$) If the primary receives a header fingerprint from a another primary, it is stored as long as the header might still be included into some learner-specific DAG. We also elide to mention that each correct validator has to check whether the received data are consistent with the local observations. primary ←primary

Header signature commitment ($\underline{\mathbf{HD}}^{*}!\boxed{\boxed{\underline{\mathbf{HD}}}}_{\sim}\rightarrow$) If all worker hashes of the received header fingerprint are known to the primary, the primary commits to the header by answering with a signature over the header. primary →primary

Header signature ($\boxed{\textbf{HD}}_p\leftarrow$) The signature of a received availability commitment is either stored or added to the aggregated signature "under construction", leading towards a certificate of availability. If the received signature completes a global quorum, it triggers the broadcasting of the completed aggregated signature, *i.e.*, the certificate of availability. primary $\leftarrow$primary

> **Availability certificate broadcast ($\boxed{\textbf{HD}}_p$!ⓐ$\Rightarrow$)** If the received commitment completes a global weak quorum for its genesis header, it broadcasts the certificate of availability. primary $\Rightarrow$primary

A (partial) execution of the availability protocol at genesis is illustrated in Fig. 2. Also, note that in some exceptional circumstances, a received header signature $\boxed{\textbf{HD}}_p$ might also need action according to the integrity protocol, as explained next.

7.2 Integrity: the general case at once

First off, the integrity-protocol re-uses the sending of signed headers $\boxed{\textbf{HD}}_p$ to the creator (from the availability-protocol) as a commitment of the signer to one unique header for the validator (and round), namely the first one signed and sent. Thus, correct validators will not sign and send any other header from the creator of the header (for the same round).

Integrity signing *Signing and sending a header to its creator implies that (a correct) primary will not sign any other header of the same creator with the same round number. (Recall that the headers at genesis implicitly are of round zero.)*

The availability protocol for primaries will use counterparts to blocks in Narwhal [DKSS22], which come with references to a quorum of blocks, namely *learner-specific blocks* and *signed block quorums*, defined as follows: a *learner-specific block* is a block header signed by a learner-specific quorum and a *signed quorum* is a quorum of blocks signed by a primary. Finally, a *typical header*, *i.e.*, one after genesis, consists of

- the identifier of a primary
- a list of worker hashes
- an availability certificate for the previous header of the same primary
- the round number
- a non-empty list of signed quorums, at most one per learner.

The first two items were already present in genesis block headers while the remaining three only come into play in later rounds. With these definitions in place, we can describe the primary actions in the availability protocol.

Header signature ($\boxed{\textbf{HD}}_p\leftarrow$) If the received signature of a header (interpreted as integrity commitment) completes a full learner-specific quorum of signatures, the received signature triggers the broadcast of one (or several) learner-specific blocks. primary $\leftarrow$primary

> **Block broadcast** ($\boxed{\textbf{HD}}_p$**!**(bk $\Rightarrow)^+$) If the received header signature completes a *learner-specific* block, for each such new block, the signature aggregator will broadcast a block to all primaries that belong to some quorum of the respective learner.[12] The (learner-specific) round number of a (learner-specific) block is derived from the block header that it is based on: it defaults to zero, unless there is a signed quorum that is the last one for the respective learner in the chain of block headers of the same primary and then it is one plus the signed quorum's round number.[13] primary $\Rightarrow$ primary

Note that there is no conceptual difference between the integrity protocol at genesis, compared to the typical case. The only difference is that headers need to carry additional information. Now, we can finish the description of heterogeneous Narwhal, by filling in the missing detail of the availability-protocol after genesis (see also Fig. 4, for the difference between headers at genesis and afterwards).

7.3 Availability after genesis

The only additional message that we have to respond to are blocks. All other responses to messages are verbatim the same as at genesis.

Block reception (bk $\leftarrow$) If a received learner-specific blocks completes a quorum of block (for the same learner) and they all carry the same round number, the primary signs and broadcasts the quorum *unless* a signed quorum of a blocks of a higher round number has been broadcast already. primary $\leftarrow$primary

> **Signed quorum broadcast** (bk**!**q$\Rightarrow$) Under the stated conditions, the primary signs a list of quorums (in the order that they were received) and broadcasts them to all primaries. primary $\Rightarrow$primary

> **Header announcement** ((bk/**WH**/a)**!**$\boxed{\textbf{HD}}^*\Rightarrow$) If the primary has already enough worker hashes for a new block header, it has an availability certificate of its previous header , and the produced signed quorum the first one it knows of, a new block header is announced by broadcasting its fingerprint. After genesis, a header carries two additional data items, namely
>
> - the availability certificate of the previous header (collected by the same primary)

[12]In other words, the broadcast is to all primaries in the union of all quorums of the respective learner.

[13]Signed quorums will be used as references to previous blocks in the learner-specific DAGs, as detailed in Section/Appendix**??**.

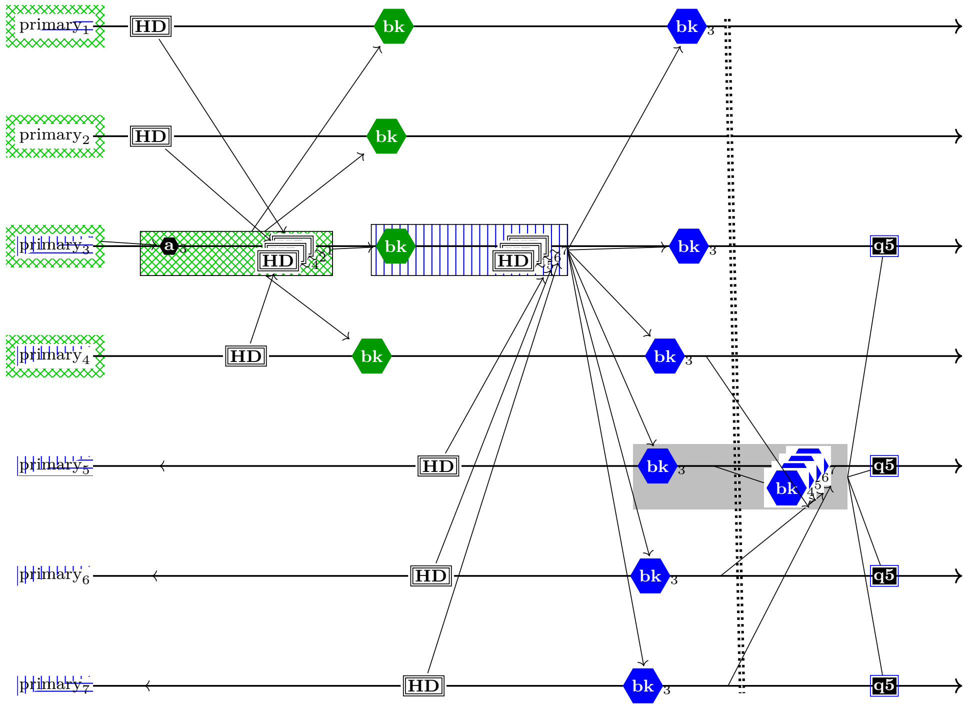

Figure 3: The integrity protocol (on the left) and the availability protocol (on the right)

- a non-empty list of hashes of signed quorums sent by the same primary, but at most one per learner
- the primary's round number, obtained by incrementing the round number of the block header that this referenced in the availability certificate.

Alternative triggers are

- a (first) worker hash arriving, in which case the header announcement includes the whole list of all signed quorums (of maximal learner-specific heights)
- the availability certificate being the missing data, which then leads to all signed quorums (of maximal learner-specific heights), and as many worker hashes as possible.

7.4 Summary

The availability protocol in a non-genesis round only differs in having

1. the additional requirement that each block header also includes the certificate of availability for the previous header of the same validator and

2. the sending and checking of signed quorums (each of which implements the reference to blocks from the previous round—in a learner-specific DAG).

As a consequence, casting an availability vote / sending a commitment message becomes a recursive commitment to storing all blocks until genesis (or the last block that some of learners might still want availabl).

8 Data structures

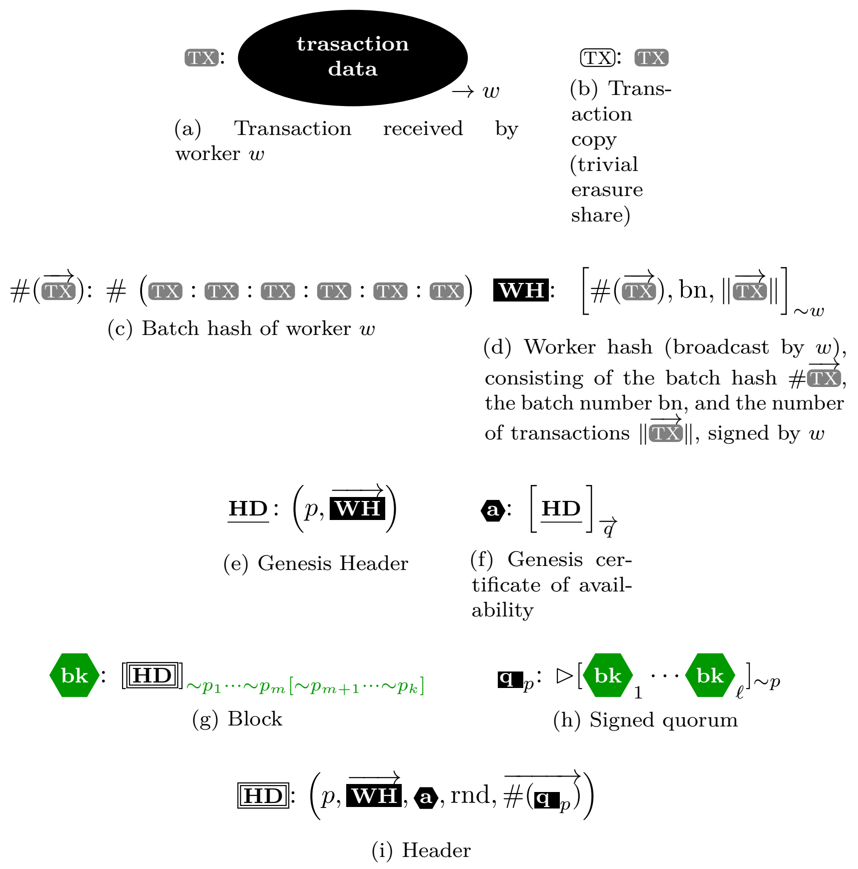
TX:
trasaction data
→ w
(a) Transaction received by worker w
TX: TX
(b) Transaction copy (trivial erasure share)
#(TX): # (TX : TX : TX : TX : TX : TX)
(c) Batch hash of worker w
WH: [#(TX), bn, ‖TX‖]~w
(d) Worker hash (broadcast by w), consisting of the batch hash #TX, the batch number bn, and the number of transactions ‖TX‖, signed by w
HD: (p, WH)
(e) Genesis Header
a: [HD]q
(f) Genesis certificate of availability
bk: [HD]~p1···~pm[~pm+1···~pk]
(g) Block
q p: ▷[bk 1 ··· bk ℓ]~p
(h) Signed quorum
HD: (p, WH, a, rnd, #(q p))
(i) Header

Figure 4: Overview of data structures

9 Transaction life cycle

A transaction goes through the following stages, first on the worker, then on the primary, finally off to execution. In more detail, on a “sunny day”, the progress towards execution, step by step, is as follows:

- submission to a worker, received as TX $\leftarrow$
- the transaction gets assembled into a batch that is referenced by a worker hash, which in turn is sent to the primary and received as **WH**
- the worker hash is included into a block header $\boxed{\textbf{HD}}$
- the block header is announced, received as $\boxed{\textbf{HD}}^*$
- the block header is provably available, via reception of enough header signatures $\boxed{\textbf{HD}}_p$
- the block header has reached integrity, via reception of enough header signatures $\boxed{\textbf{HD}}_p$, *i.e.*, the transaction is now referenced by a learner-specific block **bk**
- the block gets distributed
- the block gets a quorum of signatures **q**
- eventually,[14] the block gets commited,
  - either if the signed quorum makes it into a leader block,
  - or by indirect reference (see **??** for more detail)
- the block containing the transaction gets executed.

10 Comparison with Narwhal

Let us see how the special case of a single learner amounts to a slightly more complicated version of Narwhal. For workers nothing really chances. Note that due to the restriction that a new block header has to contain at least one signed quorum (and at most one), this means that every new header must contain a signed quorum; moreover, the signed quorum is a quorum of signatures for a previous block of the unique learner.

References

[CNG21] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483, 2021.

[DKSS22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022.

[14] In fact, this is the crucial point (see also **??**).

[SWvRM21] Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous Paxos. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, volume 184 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.