# Dispatch Protocol: Introduction to DAPoS

Dispatch Labs

April 20, 2018

**Abstract**

In this paper, we present a novel consensus algorithm, *Delegated Asynchronous Proof-of-Stake* (DAPoS), with the goal of designing a distributed state consensus algorithm that could maximize parallelizable transaction throughput. DAPoS maximizes scalability of transaction throughput by minimizing the Delegates codependency. Once transaction information is evenly distributed between Delegates, each Delegate autonomously and deterministically accepts the Transaction into their chain and reports the validity of the Transaction. Work done by Delegates is redundant for the Byzantine security of the network. DAPoS Delegates are elected by Stakeholders based on stake-weighted voting, and gossip to one another about which Transactions they have received from External Actors using cryptographically secure ECDSA signatures. Once a Delegate receives $\lfloor 2N_D/3 \rfloor$ of Delegate signatures for a given Transaction within maximum Lag Threshold, the Transaction is accepted and added to that Delegates Ledger. The validity of the Transaction is reported back to Bookkeepers, so Delegates can be evaluated and held accountable by Stakeholders.

## 1. Introduction

Cryptocurrency consensus algorithms must be tolerant to Sybil attacks [1] and Byzantine fault-tolerant to malicious or unreliable nodes [3], all while remaining as open and decentralized as possible. One of the most popular blockchain consensus algorithms is the Proof-of-Work (PoW) protocol defined in Satoshi Nakamotos Bitcoin whitepaper [2] and implemented in other popular networks like Ethereum [4]. PoW has low transaction throughput and is very computationally expensive at scale, but that cost is directly correlated with the security of the network. Delegated Proof-of-Stake (DPoS) consensus [5] dramatically increases throughput by designing a system where miners collaborate to discover blocks instead of competing. DAPoS implements a similar system of Election Cycles where Stakeholders can elect consensus leaders to reduce the overhead of replicated work without sacrificing the decentralization of the network. DAPoS differs though in that its transactions are not grouped into blocks but individually gossipped.

Gossip protocols are an alternative consensus mechanism used frequently in distributed systems before the invention of blockchain [6]. Gossip protocols have gained popularity in cryptocurrencies such as Hyperledger [7], Hashgraph [8], and Stellar [9]. Gossip protocols are proven to be fail-stop tolerant with $\lceil (N_D + 1)/2 \rceil$ good nodes and Byzantine fault-tolerant (against malicious actors) with $\lfloor (2N_D + 1)/3 \rfloor$ good nodes [10], where $N_D$ is the number of Delegates. DAPoS Delegates gossip to achieve even distribution of information before deterministically accepting and validating transactions, comparable to the replicated work of the more traditional Practical Byzantine Fault Tolerance (PBFT) [11] and Paxos [12] consensus algorithms. DAPoS applies cryptocurrency based governance to established, high-throughput consensus models to achieve scalability without sacrificing decentralization.

# 2. Definitions

## 2.1. Variables

- **Transaction**: a Transaction, $T$, is a request to update the system state. A Transaction may include many fields, but must include a timestamp ($T_{ts}$), in UTC format, with nanosecond precision (padding added as needed), as well as a Transaction Hash (formally, $T_h$) - which is used as a UID for the Transaction, and a set of cryptographically secure signatures, $T_s$.
- **Signature**: a Signature, $S[D^{(k)}]$, is created when a Delegate receives information about a specific $T$. A Signature is a byte array that can be used to recover the creating Delegates address [13]. External Actors sign their Transaction data, and upon receipt of a Transaction, Delegates add their timestamped Signature. Additionally - as the value of the byte array - each Delegate Signature contains a timestamp of when it was created, $S[D^{(k)}]_{ts}$.
- **Lag Threshold, Initial** ($L_i$): is the maximum amount of time that may have elapsed between $T_{ts}$ and the first receipt by a Delegate.
- **Lag Threshold, Gossip** ($L_g$): is the maximum amount of time that is allowed between one Delegates Signature and the next. The Lag Threshold for Delegate Gossip is critical to ensure the eventual finality of a transaction
- **Ledger**: a Delegate's "Ledger" is a record of all the Transactions that have been accepted (not just received) by that Delegate. The Ledger has an ever-sliding window at its tail wherein the Transactions have only been provisionally executed. This window bounds the worst case scenario of malicious Delegate collusion. The window size is given by:

$$w = L_i + \left( \lceil \frac{2N_D}{3} \rceil - 1 \right) L_g \tag{1}$$

## 2.2. Actors

- **Delegate**: A "Delegate" (formally, $D^{(k)}$ - where $(k \in \kappa)$ denotes a nonlinear identifier of a specific Delegate node in the set of identifier $\kappa$) is an elected computer system that is responsible for verifying $T$. A Delegate must have an accurate clock (e.g.; NTP [14]) as well as an efficient key-value storage mechanism. The set of delegates in the system is defined as $\mathcal{D}$, and has cardinality $|\mathcal{D}| = N_D$, with the constraint $N_D \geq 3$.
- **Bookkeepers**:In this system, Bookkeepers are responsible for comparing Delegate Ledgers. Bookkeepers are analogous to Learners in systems like Paxos [12]. Anyone can be a Bookkeeper and the more Bookkeepers there are, the more secure the system. Delegates send their validated transactions to Bookkeepers who in turn make relative Delegate data available for Stakeholders.

- **Stakeholders**: Stakeholders evaluate the performance of the Delegates and assigns their voting power accordingly. If the Bookkeepers continually report a Delegate's bad behavior (intentionally or unintentionally), it is the responsibility of the Stakeholders to reassign their votes to another potential Delegate.
- **External Actor**: it is a piece of software that an end user (likely a person) interacts with. An External Actor should have a reasonably accurate clock. An External Actor submits system-state update requests to Delegates in the form of Transactions.

### 2.3. Elections

In this system, Delegates are elected by the Stakeholders (those with an investment in the system) [5]. The manner and method of election is largely irrelevant to the process of reaching consensus and is out of scope for this document. However, the end result is that all nodes in the system know who their Delegates are, and Delegates can communicate with each other in a low-overhead manner (UDP, websockets, etc...). In the examples included, the number of Delegates described is the minimum number in order to illustrate the functionality of the DAPoS algorithm.

### 2.4. Transaction States

A Transaction will pass through three distinct phases, relative to each Delegate, defined by how each phase ends. These phases are not explicitly recorded on the Transaction record, but are defined here as a means of grouping Transactions for discussion purposes.

- **Received**: A Transaction is received when it has at least one Delegate signature but less than $\lfloor(2N_D+1)/3\rfloor$ within the allotted $L_g$ time frame. A transaction can stay in the received state indefinitely, and it will never be added to the Ledger (formally, $A_r$).
  The delegate state transition function $\mathbf{\Omega}$ updates the state $\delta_t^{(k)}$ of the Delegate $D^{(k)}$, after it receives a set of transactions ($B = \{T_0, T_1, ...\}$):

$$\delta_{t+1}^{(k)} \equiv \mathbf{\Omega}(\delta_t^{(k)}, B, \Sigma) \tag{2}$$

  where $|B| \geq 1$. $\Sigma$ is the set of signatures and timestamps collected by each transactions of $B$, prior to visiting $D^{(k)}$. Note that $\Sigma = \varnothing$, if $D^{(k)}$ receives the transaction directly from the user.
- **Accepted**: A Delegate will *accept* a Transaction when it sees signatures from $\lfloor(2N_D+1)/3\rfloor$ Delegates within the allotted $L_g$ time frame. This, then, would happen at or before the lapse time set by:

$$A_t = \lfloor\frac{2N_D+1}{3}\rfloor \times L_g + L_i \tag{3}$$

- **Validated**: A Delegate will *validate* a transaction once it is accepted. Validating a transaction applies the updates associated with the transaction into the system state at time $T_{ts}$. This would happen at or before $A_t$. Once a Delegate recognizes a Transaction as having been processed, responsibility of reporting the Transaction validity to Stakeholders will move on to the Bookkeepers.
  The new state $\sigma_{t+1}$ of the Ledger is given by the state transition function $\mathbf{\Lambda}$:

$$\sigma_{t+1} \equiv \mathbf{\Lambda}(\sigma_t, B^*) \tag{4}$$

  where $B^*$ is the set of validated transactions, with $B^* \subseteq B$ and $B^* \neq \varnothing$.

## 3. Example of a Transaction Flow

In this section, we outline a possible transaction flow for a single active Transaction in a system where there are no faults. In this example, the system comprises a total of three Delegates ($N_D = 3$).

1. External Actor creates $T$ with relevant data (including $T_{ts}$), sending it to $D^{(1)}$.

2. $D^{(1)}$ checks to be sure that $L_i$ has not been surpassed, using its current time and $T_{ts}$. If this does not pass or $D^{(1)}$ can determine that $T$ is obviously invalid, $D^{(1)}$ would respond synchronously to the External Actor that $T$ has been declined.

3. $D^{(1)}$ creates $S[D^{(1)}]$, pushes it into a new array, and adds the array to $T$ (formally, $T_s$).

4. $D^{(1)}$ writes $T$ into storage, using $T_h$ as the key.

5. $D^{(1)}$ selects a Delegate to visit with, in this case randomly selecting $D^{(2)}$.

6. $D^{(1)}$ sends $D_2$ a key/value map of all the Transaction Signatures it knows about in storage. The key of the map is $T_h$ while the value is $T_s$.

7. $D^{(2)}$ compares each of the $T_s$ lists provided with what it has in storage to determine which are the same, which $D^{(2)}$ has seen but have new signatures in the provided list, and which $D^{(2)}$ has never seen.

   (a) For any $T_s$ lists that are the same, $D^{(2)}$ takes no action.

   (b) For any $T_s$ lists that have new Signatures, $D^{(2)}$ should merge its stored $T_s$ list with the incoming $T_s$ list, keeping only a single unique $S[D_x]$ ($a$ "set"). In the case of two (or more) Signatures from the same Delegate with different $S[D^{(k)}x]ts$ values, all Signatures from that Delegate should be removed from the stored $T_s$ list.

   (c) For any $T_s$ lists which have never been seen, $D^{(2)}$ should include the $T_h$ in a response to $D^{(1)}$.

8. $D^{(1)}$ receives the response of $T_h$ list that $D_2$ has never seen and sends another message to $D^{(2)}$ containing the $T$ for each of those $T_h$.

9. $D^{(2)}$ receives the message and performs the following for each $T$:

   (a) $D_2$ creates $S[D^{(2)}]$; appending it to the end of $T_s$.

   (b) $D^{(2)}$ writes $T$ into storage, using $T_h$ as the key.

   (c) $D^{(2)}$ examines the resulting length of the array ($T_s.ln$) and attempts to determine if it is long enough to qualify for having been seen by at least 66% of the Delegates: $T_s.ln \geq \lceil (2N_D + 1)/3 \rceil$.

   (d) If the array is not long enough, the thread terminates.

   (e) If the array is long enough, $D^{(2)}$ will confirm that the time between the first Signature and $T_{ts}$ is equal to or less than $L_i$, formally: $(T_s[0]_{ts} - T_{ts}) \leq L_i$.

   (f) $D^{(2)}$ will then work upwords through the $T_s$ array, checking that each Signature's time differential is equal to or less than $L_g$: $(T_s[x+1]ts - T_s[x]ts) \leq L_g$.

   (g) If any faults are found, $T$ will be rejected and the thread will terminate.

   (h) When no faults are found in $\lceil (2N_D + 1)/3 \rceil$ of the differentials, $D^{(2)}$ accepts $T$ and removes it from storage.

   (i) $D^{(2)}$ will add $T$ into a list of accepted Transactions (formally, $A_l$) and the thread terminates.

10. Once all threads for the message are complete, $D^{(2)}$ will broadcast $A_l$ to every other Delegate, which in our example are $D^{(1)}$ and $D^{(3)}$.

11. $D^{(1)}$ and $D^{(3)}$ each receive the broadcast and - independently - for each $T$ included performs the following:

(a) $D^{(k)}$ seeks to accept $T$ (steps 9.c through 9.h above). $D^{(k)}$ will also check the time between when the final Signature was added ($T_s[T_s.ln]_{ts}$) and the time when the broadcast was received ($b_{ts}$) to ensure that its less than $\mathrm{L}_g$, formally: $(T_s[T_s.ln]_{ts} - b_{ts}) \leq L_g$.

(b) If $D^{(k)}$ also accepts $T$, then $D^{(k)}$ removes $T$ from its storage and the thread terminates.

(c) If $D^{(k)}$ fails to accept $T$ (and/or the final lag check fails to pass), then $D^{(k)}$ leaves $T$ in storage (to be included in future visits with other Delegates).

12. $D^{(2)}$ selects a Delegate to visit with (return to step 5).

## 4. Gossip Peer Selection

When determining which other Delegate to gossip with (or to *visit*), there is only one rule: a Delegate must visit all other Delegates before repeating any specific Delegate. Each time a Delegate has visited all the other Delegates once, they would be said to have completed a "gossip cycle", and started a new one. Gossip cycles between Delegates need not be synchronized. Beyond that rule, a Delegate may wish to attempt to be as efficient as possible - where efficiency is defined as keeping Delegate Transaction state as uniform as possible. In the abstract, this would involve tallying the total number of shared Transactions for each other Delegate available in the gossip cycle, and selecting the lowest count (with a randomized tie-breaker).

### 4.1. Cosine optimization Algorithm

The Gossip Peer selection can be formulated as a *cosine optimization*, in which the cosine must be minimized. For each Delegate $D^{(k)} \in \mathcal{D}$, we define a vector $\mu^{(k)}$ of size $|B|$, the number of transactions in $B$. The value of each element of the vector, $\mu_i^{(k)}$, is set by the status of a transaction $T_i$ ($\in B$) with respect to the Delegate, that is to say:

$$\mu_i^{(k)} = \begin{cases} 1, \text{ if the transaction } T_i \text{ has visited the Delegate } D^{(k)} \\ 0, \text{ otherwise} \end{cases}$$

The Delegate or set of Delegates, $\mathcal{D}^*$, that will gain the most information from Delegate $D^{(k)} \in \mathcal{D}$ is given by:

$$\mathcal{D}^* = \underset{j \in \kappa,\ j \neq k}{\operatorname{argmin}} \left( (\mu^{(k)})^\intercal \cdot \mu^{(j)} \right) \tag{5}$$

### 4.2. Example

In order to illustrate the selection process, we consider the case of a system with the set of delegates $\mathcal{D} = \{D^{(0)}, D^{(1)}, D^{(2)}, D^{(3)}, D^{(4)}\}$ , and at the start of a new gossip cycle. In that cycle, $D^{(0)}$ knows about $T_1$ - which has been seen by $D^{(1)}$ already (formally, $T_1[D^{(1)}]$) - as well as $T_2[D^{(1)}, D^{(2)}]$ and $T_3[D^{(2)}, D^{(4)}]$. Tallying the numbers, $D^{(0)}$ and $D^{(1)}$ share knowledge of 2 Transactions (namely $T_1$ and $T_2$), $D^{(0)}$ and $D^{(2)}$ of 2, $D^{(0)}$ and $D^{(3)}$ of 0, and $D^{(0)}$ and $D^{(4)}$ of 1. This can be more clearly represented in the form of a matrix $M$, where the columns represent the different Delegates of the system, and the rows the transactions:

$$M = \begin{array}{c} \begin{matrix} D^{(0)} & D^{(1)} & D^{(2)} & D^{(3)} & D^{(4)} \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} T_1 \\ T_2 \\ T_3 \end{matrix}$$

The matrix elements can only take 2 values in $\{0, 1\}$, reflecting whether or not a transaction has been seen by a Delegate. For example, if the Delegate $D^{(j)}$ has seen the transaction $T_i$, then the matrix element $M_{ij} = 1$, or $M_{ij} = 0$ otherwise. It becomes clear that $D^{(0)}$ should select $D^{(3)}$ so that the maximum number of Transactions may be shared in the process.

Had $D^{(3)}$ also known about $T_3$ (see $M'$ below), $D^{(0)}$ would randomly select between $D^{(3)}$ and $D^{(4)}$.

$$M' = \begin{array}{c} \begin{matrix} D^{(0)} & D^{(1)} & D^{(2)} & D^{(3)} & D^{(4)} \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} T_1 \\ T_2 \\ T_3 \end{matrix}$$

Note that $D^{(0)}$s data in the first example (matrix $M$) will be out-of-date by at least one visit (potentially up to $(N_D - 1)$ visits), so it is not the guaranteed best case, but it should result in a higher percentage than a purely random selection a majority of the time. This tally data can also be cached and updated after each visit, which would prevent having to iterate over all of the keys every gossip cycle. "Ideal peer selection" could also be replaced with a random selection of Delegates available in the cycle, as it is non-critical to the consensus.

## 5. Conclusion

DAPoS is a highly scalable cryptocurrency consensus algorithm that does not sacrifice decentralization. DAPoS applies cryptocurrency based governance to more established consensus models. Electing Delegates based on stake-weighted voting keeps the power in the hands of the network Stakeholders, and allows for a model where validators are collaborating instead of competing. Delegates use a gossip protocol to keep Transaction information evenly distributed. Delegates listen for $\lfloor (2N_D + 1)/3 \rfloor$ of Delegate signatures on a Transaction within the given thresholds before accepting a Transaction into their own chain. Transactions can then be validated and deterministically shared with Bookkeepers. Because the Bookkeepers help Stakeholders keep Delegates accountable for their performance, DAPoS operates with remarkably high levels of Scalability.

However, DAPoS still shows a number of weaknesses to be improved upon. Because Transactions are inserted into the list based on time signatures, it is likely a Transaction could be inserted before the end of a list, and the subsequent Transactions will have to be revalidated. It also implies that malicious delegates could intentionally hold Transactions to slow down the network. We have implemented the $L_g$ threshold to bound the effect of Transaction withholding, but it does mean that External Actors might need to wait the entirety of the Ledger Window to ensure a Transactions validity. The reduction of redundant computation in the network also makes it a likely target for distributed denial of service (DDoS) attacks. This can be mitigated by rotating the delegates more frequently, or implementing a random $m$ of $n$ delegate selection process to make target attacks less effective.

## References

[1] John R. Douceur, The Sybil Attack, *International Workshop on Peer-to-Peer Systems* (2002), pp. 251-260.

[2] Satoshi Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf.

[3] Leslie Lamport, Robert Shostak and Marshall Peaset, The Byzantine Generals Problem, *ACM Transactions on Programming Languages and Systems* **Vol. 4**, No. 3 (1982), pp. 382-401.

[4] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform (2013), https://github.com/ethereum/wiki/wiki/White-Paper

[5] Delegated Proof-of-Stake Consensus, https://bitshares.org/technology/delegated-proof-of-stake-consensus/

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart and Doug Terry, Epidemic algorithms for replicated database maintenance, *PODC '87 Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (1987), pp. 1-12.

[7] Hyperledger Architecture, **Vol. 1** (2017), https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf

[8] Leemon Baird, The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance (2016), https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[9] David Mazirez, The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, *Stellar Development Foundation* (2016), https://www.stellar.org/papers/stellar-consensus-protocol.pdf.

[10] Gabriel Bracha and Sam Toueg, Asynchronous Consensus and Broadcast Protocols, *Journal of the ACM*, **Vol. 32**, Issue 4 (1985), pp. 824-840.

[11] Miguel Castro and Barbara Liskov, Practical byzantine fault tolerance, *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999), pp.173186.

[12] Leslie Lamport, The Part-Time Parliament *ACM Transactions on Computer Systems*, **Vol. 16**, Issue 2 (1999), pp. 133-169.

[13] Don Johnson, Alfred Menezes and Scott Van-Stone, The Elliptic Curve Digital Signature Algorithm (ECDSA), *International Journal of Information Security*, **Vol. 1**, Issue 1 (2001), pp. 36-63.

[14] David L. Mills, Internet Time Synchronization: The Network Time Protocol, *I.E.E.E. Trans. Comm.* **39**, No 10 (1991), pp. 1482-1493.