TONSeer
A new vision for creating modern oracles based on the asynchronous TON blockchain

by Dr. Emanuele Costa, Roman Nguyen and Boris Pimonenko

12 June 2023

Contents

1 About **2**

2 Introduction **4**

3 Modern Oracle System **5**

4 Technical Concept **6**
4.1 Architecture 6
4.2 Off-chain Components 7
4.2.1 Node-based Architecture 7
4.2.2 Node Connection to Data Providers 7
4.2.3 Oracle Based Consensus 7
4.2.4 Price Aggregation 10
4.2.5 Network Mechanism 12
4.2.6 Node Rewards and Slashing 16
4.3 On-Chain Components 16
4.3.1 DataPoint and DataTable Smart Contracts 16
4.3.2 Non-cacheable Requests 17
4.4 Integration of Components 18
4.4.1 Data not cached 19
4.4.2 Data cached 20
4.4.3 Data frozen 21
4.4.4 Custom data 22

5 Advancing TONSeer's Capabilities **24**

6 Real-world Design Considerations **25**

7 Conclusion **26**

1 About

This document describes a modern approach to building an oracle system, hereinafter referred to as the TONSeer concept, in an asynchronous blockchain. The document covers the technical approach in detail, including its architecture, node-based design, consensus mechanism, data aggregation, and on-chain components. It also highlights the advantages of TONSeer, such as its focus on accuracy, reliability, cost efficiency, and speed, and its integration with the TON blockchain.

The document includes:

- A thorough analysis of the challenges associated with data delivery in the blockchain industry, including establishing trust in data sources, ensuring the accuracy of data delivery, guaranteeing data availability for smart contracts, and maintaining cost efficiency throughout the process.
- An in-depth overview of TONSeer's technical concept, which encompasses:
  - A robust architecture designed to facilitate secure and efficient data delivery.
  - A node-based design that allows for decentralized operation and enhanced reliability.
  - An innovative consensus mechanism that ensures the integrity of the data and maintains the trustless nature of the system.
  - Advanced data aggregation techniques that optimize the accuracy and reliability of the information being delivered.
  - On-chain components that enable seamless interaction between TONSeer and smart contracts.
- A comprehensive discussion of the advantages of TONSeer, including:
  - **Accuracy:** TONSeer's commitment to delivering precise and timely data makes it a reliable solution for smart contracts that require access to trustworthy information.
  - **Reliability:** The combination of TONSeer's architecture, node-based design, and innovative consensus mechanism guarantees the dependability of the data delivered to smart contracts.
  - **Cost efficiency:** By utilizing caching, advanced data aggregation, and on-chain components, TONSeer minimizes the number of transactions and overhead costs associated with data delivery, resulting in a cost-effective solution.
  - **Speed:** TONSeer's ability to deliver data promptly makes it a fast and efficient solution for smart contracts.
  - **Integration with the TON blockchain:** TONSeer leverages the full capabilities and innovations of the TON blockchain, resulting in a powerful and comprehensive oracle solution.

This document does not include:

- A detailed analysis and design of staking, collateral, slashing, and reward mechanisms for oracle nodes.
- A payment system for data transactions within the TONSeer ecosystem.
- A process for adding new data providers and creating data sets.

Despite these limitations, the document offers a thorough understanding of TONSeer's technical concept, its key advantages, and its potential to revolutionize the way oracles operate within the blockchain.

2 Introduction

The blockchain industry has seen a variety of oracle implementations, each grappling with the inherent challenges of delivering data to the blockchain. Data delivery is a complex task that demands a solution to multiple issues such as trust in data sources, accuracy of data delivery, availability of data for smart contracts, and cost efficiency. Although various implementations address these problems by developing decentralized networks with consensus mechanisms or prioritizing specific issues, the similarity of these problems with the blockchain trilemma makes a comprehensive solution difficult to achieve.

By 2023, the landscape of data provision has undergone significant transformation. Data providers now prioritize accurate data delivery and employ engineers to ensure correctness. This shift has mitigated trust issues and redirected focus towards optimization, cost efficiency, speed of data transmission, and guaranteed data delivery. With the repercussions of providing inaccurate data fully understood, data providers and market players strive to streamline the process. The result is a system that is more efficient, faster, and reliable, ensuring timely and accurate data delivery to smart contracts and fostering trust.

One of the key challenges with modern oracles is optimization. Oracle operations typically involve user smart contracts issuing separate requests for each piece of data, which are processed off-chain and returned to the contract. However, this approach generates significant system overhead that can be avoided when multiple client smart contracts request identical data. Therefore, there is a crucial need for an oracle system capable of processing identical requests while bypassing the full data extraction and verification cycle for the same query from different clients.

Existing oracle implementations tend to be expensive, necessitating the development of cost-effective, faster, and more reliable solutions. An optimized oracle system would minimize redundant data retrieval and verification processes, thus reducing costs and enhancing overall performance. This approach would streamline data delivery, ensuring that the system can handle a high volume of identical requests without compromising accuracy, security, or performance.

3 Modern Oracle System

TONSeer, an innovative oracle solution for the TON blockchain, is designed to address the complexities associated with data delivery. By optimizing the handling of identical requests, TONSeer achieves significant cost savings and efficiency through its innovative consensus mechanism and advanced data aggregation algorithms. Through data caching, the system minimizes the number of transactions and overhead costs, particularly when various client smart contracts request the same information, leading to substantial savings in overheads.

Leveraging the full potential of the TON blockchain, TONSeer is built upon one of the world's most technologically advanced platforms. TON has contributed several groundbreaking innovations, including the Infinite Sharding Paradigm, the TON Virtual Machine, and TON Storage. TONSeer provides access to these cutting-edge features, showcasing the comprehensive capabilities of TON.

Moreover, TONSeer adheres to the principles of secure construction of data delivery chains in the blockchain, following established solutions. It incorporates trustless technologies for delivering data from real-world sources and other blockchains. These features ensure fail-proof data delivery to smart contracts, thus creating a reliable foundation for economic logic.

In conclusion, TONSeer offers a modern oracle solution capable of addressing the challenges of data delivery in the blockchain industry. With its focus on accuracy, reliability, cost efficiency, and speed, TONSeer is poised to transform the way oracles operate within the blockchain ecosystem.

4 Technical Concept

The TONSeer system combines the best features of existing oracle networks with cutting-edge advancements in cryptography and optimization technologies, such as Boneh-Lynn-Shacham (BLS) signatures, to ensure data integrity and system performance. It employs the innovative Oracle Based Consensus mechanism, advanced data aggregation algorithms, and efficient caching strategies to streamline data processing and delivery. This comprehensive approach addresses the challenges faced by current oracle solutions in areas such as data accuracy, cost efficiency, scalability, and response times.

4.1 Architecture

The TONSeer architecture is designed to provide a seamless integration of on-chain and off-chain components, optimizing data retrieval and delivery. The main components of the system include the Client Smart Contract, DataTable, DataPoint, Oracle Nodes, and External Data Sources.

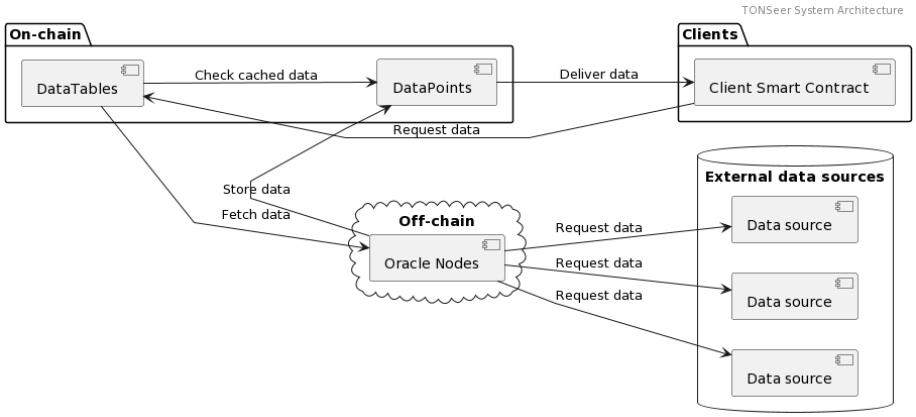

Figure 1: High-level overview of the TONSeer architecture

- **Client Smart Contract:** Represents the client that initiates data requests to the TONSeer system.
- **DataTable:** An on-chain component that groups DataPoints and helps optimize data search and retrieval.
- **DataPoint:** On-chain storage for individual data points or records, which can be cached or live, depending on the use case.
- **Oracle Nodes:** Off-chain components responsible for fetching data from external data sources, processing it, and returning the results to the Router.

- **External Data Sources:** The databases or APIs where the requested data is originally stored and accessed by the Oracle Nodes.

This architecture diagram provides a high-level overview of the TONSeer system, highlighting the integration of on-chain and off-chain components for efficient data retrieval and delivery.

4.2 Off-chain Components

The off-chain components of TONSeer play a crucial role in fetching data from external sources, processing it, and delivering it to the blockchain. Two key aspects of the off-chain components are the node-based architecture and the connection to data providers.

4.2.1 Node-based Architecture

TONSeer utilizes a node-based architecture to optimize data processing, verification, and delivery to the blockchain. This decentralized approach allows multiple nodes to work in tandem, ensuring efficient and secure data handling while maintaining the decentralized nature of the system. Each node is responsible for processing and responding to data requests from smart contracts and dApps, ensuring accurate and reliable data delivery. The node-based architecture also enhances fault tolerance, as the failure or compromise of a single node does not disrupt the entire network's functionality.

4.2.2 Node Connection to Data Providers

Each node in the TONSeer oracle network establishes connections with multiple data providers, such as financial institutions, IoT devices, or weather monitoring stations. These connections enable the nodes to fetch data from various sources and verify its accuracy. When a smart contract issues a data request, the corresponding node retrieves the required information from one or more data providers. This process may involve aggregating data from different sources, applying advanced algorithms such as weighted averages or median selection, or performing outlier detection to ensure the reliability and accuracy of the data.

4.2.3 Oracle Based Consensus

In a nutshell, Oracle Based Consensus is a subset of Distributed Ledger Consensus and uses multiple oracles and a consensus algorithm to derive factual data for smart contracts. They help ensure that data passed onto the blockchain is accurate and trustworthy.

In the last decade, a plethora of consensus algorithms have been devised both for general blockchain networks and Oracle networks. In TONSeer, we propose to research and expand a simple and efficient Oracle-based consensus protocol for asynchronous byzantine systems[1] that we believe can fit particularly well also in the overall TON asynchronous architecture.

The protocol is based on failure detectors providing lists of processes suspected to be faulty. Like most consensus protocols, the one that is presented here proceeds in consecutive asynchronous rounds. The design principles are simplicity on the one hand and time and communication efficiency on the other hand.

Let us define:

- n as the total number of processes,
- f as the maximum number of processes that can be faulty,
- r as an asynchronous consecutive round within the protocol.

A *run* is a continuous execution of the protocol and a run is logically divided into a sequence of subruns where in each subrun a decision on one or messages is taken. A process is mute (with respect to a protocol) if after some time it does not send (some or all) protocol messages (with headers) it was supposed to send. As an example, a process that, during a round r, never sends a message it is assumed to send during that round, is mute.

The protocol is based on the Rotating Coordinator Mechanism[3] whereby all active processes cyclically become coordinators for one subrun thus avoiding resorting to a voting algorithm to recover from the coordinator's failures.

The proposed 3Pmute-based Byzantine consensus protocol is an adaptation to Byzantine asynchronous systems of the simple and elegant consensus protocol proposed by Berman and Garay for synchronous systems[2]. This adaptation keeps the simplicity of the original synchronous protocol but requires a stronger assumption (namely $n > 6f$ instead of $n > 4f$) to cope with asynchrony. Moreover, it has the nice property of allowing the processes to decide in a single communication step when no process is faulty and all processes propose the same value.

In the protocol, the processes proceed in consecutive asynchronous rounds. Each round is made up of two phases. The first phase of each round is a global exchange, while the second phase is based on the rotating coordinator paradigm.

In addition to Termination and Agreement, a Byzantine protocol must ensure Validity, namely, the decided value has to be "v" when all correct processes propose "v". To attain this goal, the protocol is based on the following principle:

1. If the occurrence number of the most current value bypasses some threshold, that value will be decided.
2. Otherwise, the coordinator paradigm is used to eventually force a value to be adopted by enough processes so that the previous property becomes satisfied.

This principle is implemented as follows:

```
Function Consensus(v_i)

init: est_i ← v_i; r_i ← 0;

repeat forever
(401)  r_i ← r_i + 1; V_i ← [⊥, ..., ⊥]; c ← ((r_i − 1) mod n) + 1;
       ——————————————— Step 1 of round r ———————————————
(402)  broadcast VAL(r_i, est_i);
(403)  wait until ( VAL(r_i, −) or DEC(−) messages have been received from all non-suspected
                    processes and from at least (n − f) distinct processes );
(404)  for each j: if (VAL(r_i, est_j) or DEC(est_j) received from p_j) then V_i[j] ← est_j endif;
(405)  if (∃v ≠ ⊥ : #_v(V_i) > n/2) then dominating_i ← v else dominating_i ← est_i endif;
       ——————————————— Step 2 of round r ———————————————
(406)  if i = c_r then broadcast COORD(r_i, dominating_i) endif;
(407)  if (#_{dominating_i}(V_i) ≥ n − 2f − #_⊥(V_i))
(408)              then est_i ← dominating_i;
(409)                   if (#_{dominating_i}(V_i) ≥ n − f) then broadcast DEC(est_i); return (est_i) endif
(410)              else wait until (COORD(r_i, −) or DEC(−) received from p_c or p_c is suspected);
(411)                   if (COORD(r_i, x) or DEC(x) received from p_c)
(412)                              then coord_val_i ← x
(413)                              else coord_val_i ← dominating_i
(414)                   endif;
(415)                   est_i ← coord_val_i
(416)  endif
end repeat
```

During the first phase of a round r, the processes exchange their estimates est_i of the decision value (line 402), and build their local view of current estimates (line 405). Let us notice that the *DEC(—)* messages are the decision messages to be broadcasted. Finally, each process p_i determines, and stores in $dominating_i$, the majority estimate if any (otherwise, $dominating_i$ is set to est_i). The aim of the second phase is to allow the processes to decide, or if they cannot, to adopt the same estimate value for the next round. To that end, p_i adopts, as its new estimate, the value $dominating_i$ if it is "present enough" in V_i (line 408). "Present enough" means that it appears at least $(n-2f-\#\perp(V_i))$ times. Moreover, if the value $dominating_i$ is "very present" in V_i, p_i decides it (line 409). "Very present" means that it appears at least $n-f$ times. The proof gives insight as to why these threshold values, namely, $(n-2f-\#\perp(V_i))$ and $(n-f)$, have been chosen.

Otherwise (the value $dominating_i$ is not present enough), the round-based coordinator paradigm is used to force the processes p_i to adopt the same value. More precisely, during each round, the corresponding round coordinator broadcasts its dominating value v (line 406) to try to impose it to the other processes. Each process p_i that receives it, stores it in $coord\_val_i$ (line 412). If it does not receive it or suspects the coordinator, p_i uses its current *dominating* value as a default value (line 413). Then, p_i adopts the resulting $coord\_val_i$ as its new estimate (line 415).

If p_i decides, it first broadcasts *DEC(est_i)* to provide the other processes with its last estimate. In that way, the other processes cannot be blocked in future rounds by the fact that p_i will no longer send messages. So, as soon as a process p_k receives *DEC(est_i)* from p_i at line 411 of round r, this *DEC(est_i)*

message is used as the message p_i would have sent to p_k at any round $r' \geq r$ until p_k decides.

The protocol discussed above requires only $n > 4f$, but each round is made up of four communication steps (and the earliest decision is after 4 communication steps). It is possible to devise a similar (complementary) protocol that allows the processes to decide in one communication step in favorable circumstances with $n > 6f$[1]. This option may also be researched during the development of TONSeer. Given their simplicity and efficiency, both options are suitable for on-chain computation.

In conclusion, this section assumes that the underlying system is equipped with a failure detector that is able to detect mute processes, the definition of which is based on the notion of protocol message. A protocol message is a message that could have been sent during a run of the protocol. For example, *VAL(r, v)* messages (for any value of r) and *DEC(v)* messages are the only protocol messages that can be generated during a run. In that case, *VAL(r)* (or *DEC*) is considered as the header of the message, while v is the data part of the message. The syntax of a protocol message corresponds to the syntax of a message generated by the protocol. If that message has been generated by a Byzantine process it can carry wrong data (e.g., when considering the previous protocols, the data v in *VAL(r, v)* or *DEC(v)* can be wrong).

We emphasize that a Byzantine process is not mute if, during each round, it sends all the messages with headers that would have been sent if it was correct. Of course, as it is Byzantine, the data part of those messages can be wrong. Let us observe that the set of mute processes also includes all crashed processes. To cope with mute processes, we consider the class of failure detectors, denoted *3Pmute*, that includes all the failure detectors that satisfy the following properties:

Muteness Strong Completeness: Eventually, every mute process is permanently suspected by every correct process.

Eventual Strong Accuracy: There is a time after which no correct process is suspected.

4.2.4 Price Aggregation

Price aggregation is a sub-case of data aggregation and transformation operations, but it is worth examining and implementing because it highlights many of the issues and technical decisions needed to properly transform data for Oracle processing.

When it comes to prices, there are mainly two scenarios:

1. A specific price is needed from 1 to n data sources, such as a trading bot arbitraging between two exchanges.

2. An aggregate price is needed from 1 to n data sources, such as in derivatives instruments calculation for a given commodity trading across exchanges with various degrees of confidence intervals.

The potential Byzantine behavior of the first case is handled by the on-chain consensus algorithms previously introduced, while the second case has to be processed off-chain due to the following reasons:

- The potential for exploitation during the aggregation calculations, such as front running.
- The required aggregation operations can be computationally too complex, both in terms of cost and execution time, or not available for an on-chain smart contract.

In our approach, we extend the algorithm used by Pyth[1] and make it less dependent on data source (data publisher) estimates.

The TONSeer aggregation algorithm aims to have three properties. First, it should be robust to manipulation, both accidental and intentional, by publishers. For instance, if most publishers submit a price of $100 and one publisher submits a price of $80, the aggregate price should remain close to $100 and not be overly influenced by the single outlying price. This property ensures that the Pyth price remains accurate even if a small number of publishers submit prices that are far from the market. This scenario is depicted in graph (a) in the figure below.

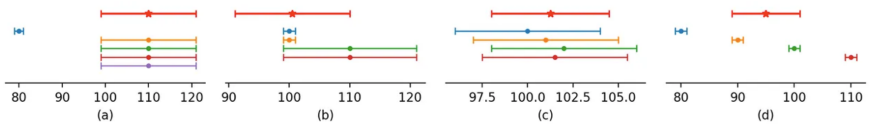

Figure 2: Graphs illustrating the price aggregation scenarios

Second, the aggregate price should appropriately weight data sources with different levels of accuracy. Pyth allows publishers to submit a confidence interval because they have varying levels of accuracy in observing the price of a product. For example, US equity exchanges have different levels of liquidity, and less liquid exchanges have wider bid/offer spreads compared to more liquid ones. This property can result in situations where one exchange reports a price of $101 ± 1, and another reports $110 ± 10. In these cases, the aggregate price should be closer to $101 than $110. This scenario is depicted in graph (b) in the figure.

Finally, the aggregate confidence interval should reflect the variation between publishers' prices. In reality, there is no single price for any given product. At any given time, every product trades at a slightly different price at various venues around the world. Furthermore, a trader will obtain a different price if they immediately buy or sell the product. We would like Pyth's confidence

[1] `https://pythnetwork.medium.com/pyth-price-aggregation-proposal-770bfb686641`

interval to reflect these variations in prices across venues. Graphs (c) and (d) in the figure depict two different cases where there are price variations across exchanges.

Pyth's aggregation algorithm follows a simple two-step process. The first step computes the aggregate price by giving each publisher three votes: one vote at their reported price and one vote at each of their reported price plus or minus their confidence interval. The algorithm then takes the median of all the votes. The second step computes the distance from the aggregate price to the 25th and 75th percentiles of the votes, and selects the larger of the two as the aggregate confidence interval.

However, the Pyth algorithm has two main issues:

1. It requires publishers to submit their confidence interval estimates, which not all publishers may have the ability to provide.

2. It is still prone to publisher manipulation or inaccuracies in estimating confidence intervals.

Instead of relying on publishers to submit their confidence interval estimates, these intervals can be obtained through methods such as simple linear regression on each publisher's reported prices.

In Linear Regression, there is a concept called a prediction interval. It involves looking at a specific value of x, denoted as x_0, and finding an interval around the predicted value $\hat{y}_0$ for x_0 such that there is a 95% probability that the real value of y (in the population) corresponding to x_0 is within this interval. This is illustrated in the graph on the right in the figure below.

$$\hat{y}_0 \pm t_{\text{crit}} \cdot s.e. \tag{1}$$

The 95% prediction interval for the forecasted value $\hat{y}_0$ for x_0 is given by:

$$s.e. = s_{y \cdot x} \sqrt{1 + \frac{1}{n} + \left(\frac{(x_0 - \overline{x})^2}{S \cdot S_x} \right)} \tag{2}$$

Additionally, there is sound scientific literature available on a range of methods for aggregating confidence interval estimates of unknown quantities [4] [5]. These methods allow for the creation of independent estimates for confidence intervals and their robust aggregation.

In our proposal for TONSeer, we aim to extend the Pyth approach and develop a more robust and less error-prone aggregation algorithm based on existing peer-reviewed literature.

4.2.5 Network Mechanism

The decentralized oracle system utilizes a robust Oracle Based Consensus mechanism with asynchronous rounds and rotating coordinators to ensure data accuracy and reliability among nodes. This process comprises several steps, including data fetching, hashing, consensus voting, and data delivery.

- **Data Fetching:** Once a node receives a request from the Router, it retrieves the relevant data from one or multiple data providers. Other nodes in the network independently obtain the same data to verify its accuracy.
- **Hashing:** Each node creates a hash of the obtained data. Hashing serves as a means to compare and verify data across nodes without exposing the actual content. It also helps to maintain the network's privacy and security.
- **Consensus Voting:** The first node initiates a voting process to reach consensus among other nodes in the network. During this process, the other nodes independently verify the data by comparing hashes. If the hashes match, the nodes sign the data using their individual signatures. The system employs an Oracle Based Consensus mechanism, with asynchronous rounds and a rotating coordinator, which allows for a more flexible and efficient consensus process.
- **Data Delivery:** Once consensus is reached, the first node combines the individual signatures using BLS (Boneh-Lynn-Shacham) signatures, which provide an efficient and secure way to aggregate signatures. The aggregated BLS signature and data are then sent back to the Router and delivered to the requesting smart contract. This final step ensures that only accurate and reliable data enters the blockchain, maintaining the integrity of the system.

By employing this consensus mechanism, the decentralized oracle network guarantees accurate data delivery, fostering trust and security in the blockchain ecosystem.

Nodes use the Gossip protocol to share data and signatures with each other. The Gossip protocol, also known as Epidemic protocol, is a communication model used in distributed systems for disseminating information among nodes in a network. The protocol operates by having each node in the network share information with its neighbors, who in turn share the information with their own neighbors, and so on. This process creates exponential growth in information dissemination, ensuring that information quickly propagates across the entire network.

In the context of TONSeer's oracle subsystem, using the Gossip protocol has several benefits. Firstly, it enables efficient and rapid dissemination of data and signatures among nodes. By sharing information with only a few neighbors, the protocol helps to reduce the load on individual nodes and minimize latency in the communication process. This is particularly important when dealing with time-sensitive data or large-scale decentralized networks.

Secondly, the Gossip protocol enhances the robustness and fault tolerance of the network. Since each node shares information with multiple neighbors, the failure of a single node or communication link does not significantly impact the

overall performance of the system. The network can continue to function and reach consensus even in the presence of faulty or malicious nodes.

Lastly, the Gossip protocol provides a level of privacy and security, as nodes do not need to reveal their entire list of neighbors or communicate directly with all other nodes in the network. This feature can help protect the network from targeted attacks and make it more resistant to potential security threats.

The provided diagram offers a visual representation of the interactions between the Router and oracle nodes, as well as the processes that occur within each oracle node, such as data verification, voting, data propagation using the Gossip protocol, asynchronous rounds, rotating coordinator, and creation of aggregated BLS signatures.

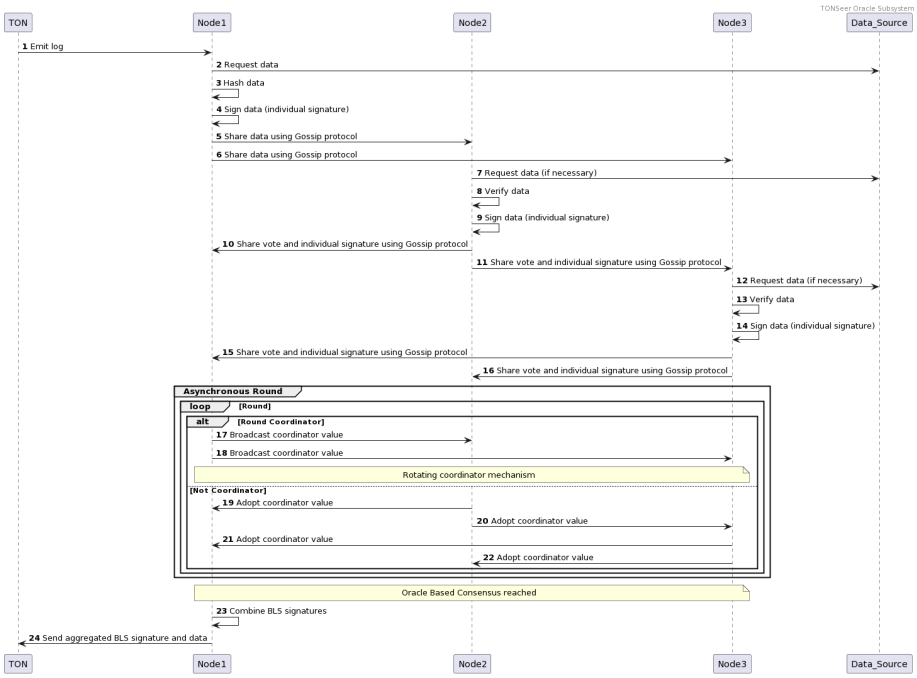

Figure 3: Oracle Consensus Process with Asynchronous Rounds and Rotating Coordinator

- **TON → Node1: Emit log**
  DataPoint initiates the process by emitting a log.
- **Node1 → DataSource: Request data**
  Node1 requests data from the data source.
- **Node1 → Node1: Hash data**
  Node1 hashes the received data.

- **Node1 → Node1: Sign data (individual signature)**
  Node1 signs the hashed data with an individual signature.
- **Node1 → Node2 and Node1 → Node3: Share data using Gossip protocol**
  Node1 shares the data and individual signature with Node2 and Node3 using the Gossip protocol.
- **Node2 and Node3 → DataSource: Request data (if necessary)**
  Node2 and Node3 request data from the data source if necessary.
- **Node2 and Node3 → Node2 and Node3: Verify data**
  Node2 and Node3 independently verify the received data.
- **Node2 and Node3 → Node2 and Node3: Sign data (individual signature)**
  Node2 and Node3 sign the verified data with their individual signatures.
- **Node2 and Node3 → Node1 and Node3, Node1 and Node2: Share vote and individual signature using Gossip protocol**
  Node2 and Node3 share their votes and individual signatures with each other and Node1 using the Gossip protocol.
- **Asynchronous Round loop**
  The system enters an asynchronous round loop, where nodes cyclically become coordinators.
- **Round Coordinator broadcasts coordinator value**
  The current round coordinator broadcasts its coordinator value to the other nodes.
- **Non-coordinator nodes adopt coordinator value**
  The nodes that are not coordinators in the current round adopt the coordinator value.
- **Oracle Based Consensus reached**
  The consensus process continues until the Oracle Based Consensus is reached among the nodes.
- **Node1 → Node1: Combine BLS signatures**
  Node1 combines the BLS signatures from all participating nodes.
- **Node1 → Router: Send aggregated BLS signature and data**
  Node1 sends the aggregated BLS signature and data back to the DataPoint to store.

4.2.6 Node Rewards and Slashing

The decentralized oracle system includes its own token, which has several purposes. Firstly, it compensates nodes for their work in processing and verifying data. Smart contracts must use this token to pay for the data they receive. The earned tokens are then distributed among the participating nodes as a reward for their contribution to maintaining the accuracy and reliability of the system. The amount of tokens a node receives as a reward might be proportional to their staked tokens, the amount of data they processed, or their overall contribution to the network.

Secondly, tokens also serve as collateral for nodes to join the network. Nodes must lock up a certain amount of tokens to become part of the network, ensuring that they have a stake in maintaining the system's integrity. If a node provides incorrect data too often or is consistently offline, a portion of their staked tokens can be slashed as a penalty. This aligns the interests of the nodes with the overall goal of the system.

4.3 On-Chain Components

4.3.1 DataPoint and DataTable Smart Contracts

In TONSeer, each individual data unit (such as a record in a database for a candlestick on March 31, 2023, for the USDT/TON pair) is represented by a separate smart contract called DataPoint. These DataPoints are then grouped together into a DataTable.

Each DataPoint contains all the required information, such as the name of the pair, the price, and the date. All data types can be divided into three groups: immutable, mutable, and indexed.

- **Immutable data** is data that is the same for two DataPoints from the same DataTable, like the name of the pair, which will always be USDT/-TON. This data is stored in the smart contract code so that the code of similar units from different DataTables may differ.
- **Mutable data** is data that is different in each DataPoint of the same DataTable but is still the data that the client is looking for. It cannot be used for search, filtering, and indexing, and is private data that is stored in the contract state and does not affect its address.
- **Indexed data** is data that is different for different DataPoints but is public, like the date or the name of the exchange. This data is passed to the contract's initState, which affects its final address, making it possible to be calculated and found on-chain.

In our example, the standard DataPoint is a smart contract that contains the pair name in the code, the date in the initState, and the price in the state.

To get data on the price for a certain date, the client smart contract only needs to calculate the DataPoint address with the necessary data by taking the

code of the required DataPoint and putting the initState with the required date there. The client can then send a message to this contract and get the data that is needed.

The client smart contract retrieves the code for the DataPoint from the DataTable, which groups DataPoints together. The DataTable stores the code of a specific type of DataPoint and performs various functions to restrict access to data, register data providers, and store data in order to optimize the search for DataPoints, such as the address of the last DataPoint if the client only needs live data. In reality, the client only needs to pass the data to the DataTable without calculating the hash to obtain the address of the desired DataPoint. Therefore, there is no need to store or request the code of the DataPoint for each call, which significantly optimizes the data retrieval operation. Thus, the DataTable is a contract that combines DataPoints, facilitates the search for the necessary DataPoints, and provides access to DataPoints.

The DataTable already solves many optimization problems. However, if the contract only requires live data or if the data itself does not require storing history, then there are two types of DataPoints available: live and constant. The live DataPoint is used to display only the latest relevant data, while the constant DataPoint is used for storing history.

It is important to note that there can only be one live DataPoint within the DataTable, which is updatable, while there can be many constant DataPoints, which have parameters for "searching".

4.3.2 Non-cacheable Requests

While caching data using DataPoint and DataTable contracts is efficient for frequently requested data, there are cases where non-cacheable requests are necessary. These requests, such as custom HTTP requests, require real-time data or unique queries that are not suitable for caching. In these scenarios, the event initiation process is employed to deliver data to client smart contracts.

For these requests, the oracle system must be flexible and able to process the data dynamically, without relying on cached data from DataPoint and DataTable contracts.

To deliver data to client smart contracts for non-cacheable requests, the oracle system leverages an event initiation process. This process works as follows:

- **Client Request:** The client smart contract sends a request to the oracle system, specifying the required data and any necessary input parameters. For custom HTTP requests, the client may include the URL, request method (GET, POST, etc.), and any additional headers or parameters needed to fetch the data.

- **Log Emission:** Upon receiving the request, the oracle system emits a log containing the details of the client request. These events are monitored by the oracle nodes, which are responsible for processing the data and returning it to the client smart contract.

- **Data Retrieval:** The oracle nodes retrieve the requested data from the appropriate data providers or external services. For custom HTTP requests, the nodes make the necessary API calls and obtain the data in real time.
- **Data Processing and Verification:** Once the data is retrieved, the nodes process and verify it according to the specified request parameters. This may involve data aggregation, filtering, or manipulation to ensure accuracy and reliability.
- **Data Delivery:** After processing and verification, the oracle nodes deliver the data back to the client smart contract through a callback function or by emitting a response event. The client smart contract then consumes the data as needed for its operation.

By employing the log initiation process for non-cacheable requests, the oracle system can dynamically handle custom HTTP requests and other unique data queries, providing real-time and accurate data to client smart contracts.

4.4 Integration of Components

The section aims to provide a comprehensive overview of the key steps and interactions involved in the data delivery process for the decentralized oracle network. The intricate interplay between the on-chain and off-chain components is crucial to delivering accurate and reliable data to clients. This analysis will examine each aspect in detail, shedding light on the complexity and robustness of the network, which forms the backbone of numerous blockchain-based applications and services.

- **Client Request Initiation:** Clients initiate requests for data through smart contracts or decentralized applications (dApps) by specifying the data type, source preferences, or additional parameters. These requests may involve querying specific data, such as exchange rates, weather conditions, or sports results, from specified data sources.
- **Request Processing:** Upon receiving a client request, DataTable determines whether to fetch the data from existing DataPoints or oracle nodes. This decision is based on factors such as data availability, freshness, and the client's preferences.
- **Data Retrieval from DataPoints:** If the data is available in an existing DataPoint, DataTable forwards the request to the corresponding DataPoint. Both cached and non-cached data scenarios are considered in this process.
  - **Cached Data:** In the "Data cached" scenario, DataTable checks if the requested data is already cached in the DataPoint. If so, the DataPoint delivers the cached data directly to the client, bypassing the need to involve oracle nodes.

  - **Data Not Cached:** In the "Data not cached" scenario, DataTable recognizes that the requested data is not cached in the DataPoint. DataTable then forwards the request to the oracle nodes to fetch the data from off-chain sources.

- **Data Retrieval from Oracle Nodes:** If the data needs to be fetched from oracle nodes, DataTable emits a log that is detected by the nodes. This log prompts the nodes to execute off-chain processes to fetch the required data. Both custom data and data freeze scenarios are addressed in this process.
  - **Custom Data:** In the "Custom data" scenario, the client requests specific data that can not be cached or available in existing DataPoints. Client directs the request to the oracle nodes, which fetch the data from off-chain sources, process and verify it, and deliver.
  - **Data Frozen:** In the "Data frozen" scenario, the requested data is temporarily unavailable due to lack on TONCOINs on DataPoint. DataTable recognizes the data is frozen, directs the request to the oracle nodes. Once the data is fetched, processed, and verified, the oracle nodes unfreeze DataPoint, which then delivers the data to the client.

- **Off-chain Data Processing:** Off-chain data processing involves several steps, including fetching data from external sources, hashing the data for integrity, consensus voting among Oracle nodes to agree on the data's validity, and data delivery to the router system.

- **Data Delivery** After reaching consensus, the oracle nodes create or unfreeze DataPoint (if it's not custom request, when oracles send data directly to client smart contract), which is responsible for delivering the data to the client after creation.

The next section provides a series of sequence diagrams that represent the most significant processes in the decentralized Oracle network. These diagrams offer a visual representation of the interactions between the on-chain and off-chain components, making it easier to understand the overall structure and flow of the data delivery process. Studying these diagrams can provide a deeper understanding of the efficiency and robustness of the decentralized oracle network and its potential to revolutionize data exchange in the blockchain ecosystem.

4.4.1 Data not cached

This sequence diagram describes the process of handling a data request in a decentralized oracle network when the requested data is not cached within the DataPoint component. It illustrates how the system fetches, processes, and delivers data while also caching the data for future use.

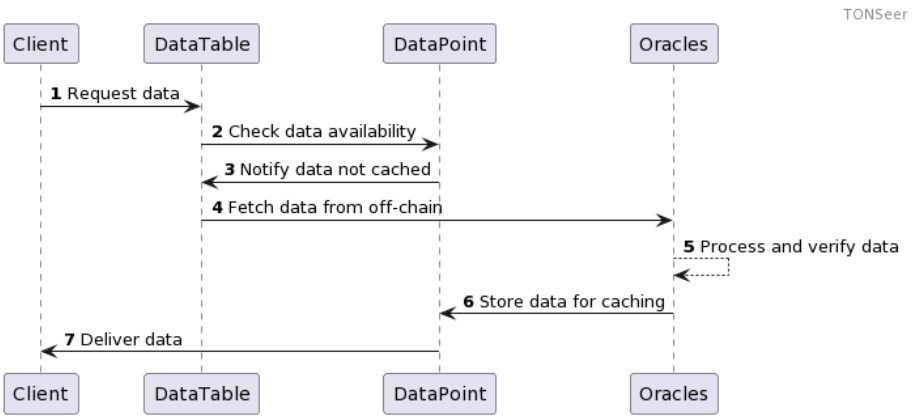

Figure 4: Data not cached

1. **Client → DataTable: Request data**
   Client initiates a request for data by sending a message to the DataTable.

2. **DataTable → DataPoint: Check data availability**
   DataTable checks the associated DataPoint to determine if the requested data is available.

3. **DataPoint → DataTable: Notify data not cached**
   DataPoint informs the DataTable that the requested data does not exist (DataPoint doesn't exists or frozen).

4. **DataTable → Oracles: Fetch data from off-chain**
   DataPoint forwards the request to the off-chain Oracles to fetch the required data.

5. **Oracles → Oracles: Process and verify data**
   Oracles process and verify the fetched data independently, ensuring its accuracy and reliability.

6. **Oracles → DataPoint: Store data for caching**
   DataPoint stores the fetched data for caching and retrieval.

7. **DataPoint → Client: Deliver data**
   DataPoint delivers the fetched data to the client.

4.4.2 Data cached

This sequence diagram describes the process of handling a data request in a decentralized oracle network when the requested data is cached within the DataTable and DataPoint components. It illustrates how the system efficiently

retrieves and delivers cached data to the client, reducing the need for off-chain data fetching and processing.

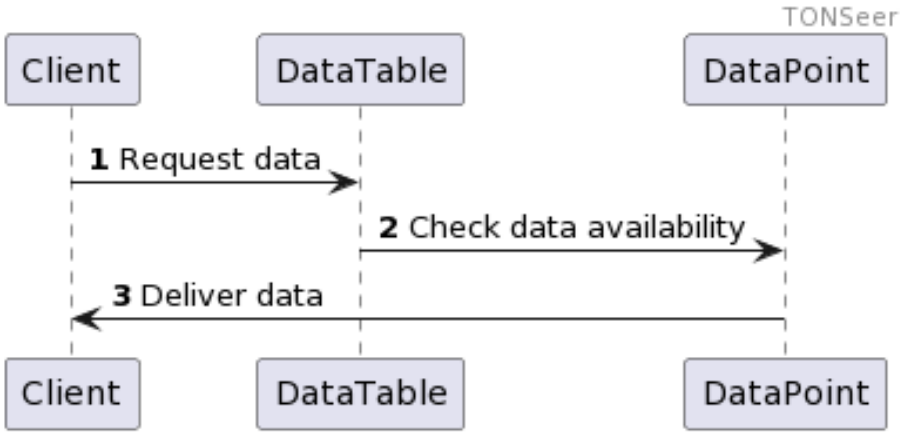

Figure 5: Data cached

1. **Client → DataTable: Request data**
   Client initiates a request for data by sending a message to the DataTable.
2. **DataTable → DataPoint: Check data availability**
   DataTable checks the associated DataPoint to determine if the requested data is available.
3. **DataPoint → Client: Deliver data**
   DataPoint delivers the cached data to the client.

4.4.3 Data frozen

This sequence diagram describes the process of handling a data freeze scenario in a decentralized oracle network, illustrating how the system can adapt and deliver the requested data even when faced with temporary data unavailability.

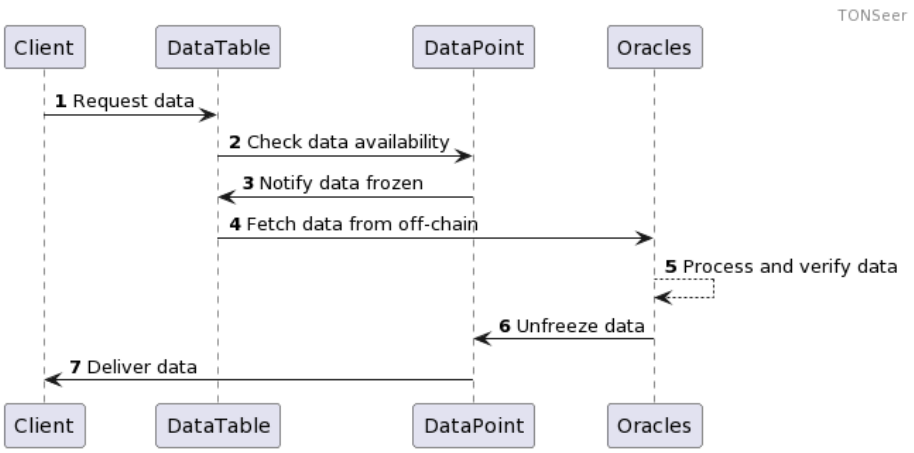

Figure 6: Data frozen

1. **Client → DataTable: Request data**
   Client initiates a request for data by sending a message to the DataTable.
2. **DataTable → DataPoint: Check data availability**
   DataTable checks the associated DataPoint to determine if the requested data is available.
3. **DataPoint → DataTable: Notify data frozen**
   DataPoint notifies DataTable that the data is frozen.
4. **DataTable → Oracles: Fetch data from off-chain**
   DataTable forwards the request to the off-chain Oracles to fetch the required data.
5. **Oracles → Oracles: Process and verify data**
   Oracles process and verify the fetched data independently, ensuring its accuracy and reliability.
6. **Oracles → DataPoint: Unfreeze data**
   Oracles unfreeze the data, making it available for future requests.
7. **DataPoint → Client: Deliver data**
   DataPoint delivers the fetched data to the client.

4.4.4 Custom data

This sequence diagram describes the process of handling a custom data request in a decentralized oracle network, illustrating how the system can fetch, process, and deliver data that is not cached within the DataTable and DataPoint components.

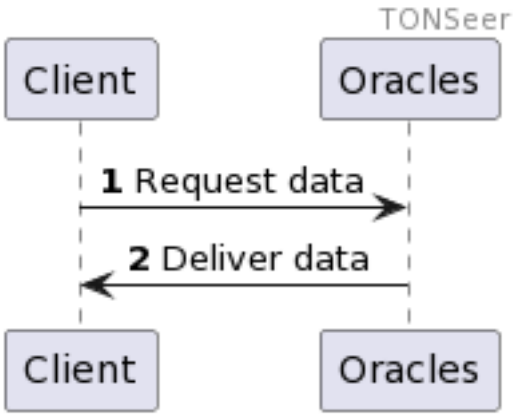

Figure 7: Custom data

- **Client → Oracles: Request data**
  Client initiates a request for data by sending a message to the Oracles.
- **Oracles → Client: Deliver data**
  Oracles directly deliver the fetched data to the client.

5 Advancing TONSeer's Capabilities

The future of TONSeer holds great potential for two significant advancements: the implementation of trustless technology for data verification and the establishment of a data marketplace within the TON ecosystem.

While these advancements offer unique and groundbreaking features, it is essential to emphasize that they should be pursued after establishing a robust and reliable foundation for the primary system. The core functionalities and principles of TONSeer should be solidified and thoroughly tested before exploring these exciting possibilities.

- **Trustless Technology:** The integration of trustless technology, particularly for data verification from other blockchains, has the potential to revolutionize the reliability and efficiency of the oracle network. By leveraging the inherent trustworthiness of data from platforms like Uniswap or Aave, TONSeer can seamlessly obtain accurate price data. One possible approach is to incorporate a lightweight Ethereum client on the TON blockchain, utilizing the SyncCommittee feature from a recent update. This lightweight client would enable TON to accept entire blocks from other blockchains and verify their authenticity within the TON network itself, eliminating the need for external validation. This innovative solution ensures data correctness and reduces reliance on trust in relayers. Effective caching becomes even more critical in this scenario, as delivering data with guaranteed accuracy may come at a higher cost, making efficient caching crucial to minimize expenses.

- **Data Marketplace:** Another exciting prospect for TONSeer is the transformation into a data marketplace, enabling individuals and organizations to supply data within the TON ecosystem. While decentralized oracle networks are essential for specific use cases, many data sources have a vested interest in providing accurate information, reducing the need for extensive validation by oracles. By creating a data marketplace, TONSeer can foster a vibrant ecosystem where data providers are responsible for the quality of their data, and clients can make informed choices based on their assessment. This data marketplace aligns with the initial vision of TON, facilitating easy access to a wide range of data across various domains, such as TON sites. By streamlining information exchange between providers and users, this marketplace can drive innovation and progress in fields like science, technology, and business.

6 Real-world Design Considerations

While the current design of TONSeer focuses on the core technical aspects of a decentralized oracle network, it is important to recognize that real-world systems involve various complexities and considerations beyond pure technicalities. To fully realize its potential and ensure long-term success, several critical areas require further research and development. This section outlines these important aspects that are essential for the system's full potential and viability.

- **Collateral, Slashing, and Node Rewards:** An integral part of a decentralized oracle network is the incentivization and security mechanisms governing node behavior. In a real-world system, it is necessary to design detailed staking, collateral, slashing, and reward mechanisms for oracle nodes. These mechanisms should encourage honest behavior, discourage malicious or negligent actors, and maintain the overall security and reliability of the network.
- **Data Payment:** A robust payment system for data transactions is vital for the success of the system. Designing a secure, efficient, and transparent payment mechanism between data providers and users is crucial. This ensures fair compensation for data providers and promotes a sustainable and thriving data marketplace within the network.
- **Adding Data Providers and Creating DataTables:** Establishing a streamlined process for adding new data providers and creating DataTables is essential for the agility and adaptability of the oracle network. Designing a user-friendly system for DataProvider onboarding and DataTable creation enables the network to stay up-to-date and responsive to the evolving demands of the data market. This process ensures the continued growth, diversity, and utility of the data marketplace within TONSeer.

By considering these future design considerations, the real-world implementation of TONSeer will be equipped to provide a reliable, decentralized, and adaptable oracle network. This will facilitate efficient and secure data exchange within the TON blockchain ecosystem and beyond, addressing both technical and real-world challenges for a comprehensive and successful system.

7 Conclusion

TONSeer is an innovative oracle system concept built on the TON blockchain that aims to address the challenges of delivering accurate and reliable data to smart contracts. With its robust architecture and efficient caching mechanism, TONSeer ensures cost-efficiency and reduces overhead costs associated with maintaining nodes and processing transactions.

One of the key advantages of TONSeer is its reliability in delivering data accurately and promptly. By adhering to secure data delivery chain principles and leveraging trustless technologies, TONSeer provides a secure and trustworthy data delivery system that can source data from real-world and blockchain-based sources.

Cost-efficiency is another significant benefit of TONSeer. By leveraging the high-speed and cost-effective infrastructure of the TON blockchain, TONSeer delivers data at a lower cost compared to traditional oracle systems. The caching mechanism further reduces the number of transactions and overhead costs when multiple client smart contracts request the same data, resulting in substantial cost savings for users.

Furthermore, TONSeer's advanced architecture enables it to handle a wide range of data types, leveraging the capabilities of the TON blockchain. With features such as the Infinite Sharding Paradigm, the TON Virtual Machine, and TON Storage, users can experience the full potential of the TON blockchain and benefit from its groundbreaking innovations.

In conclusion, TONSeer is a cutting-edge oracle system concept that offers reliability, cost-efficiency, and fast data transfer. It has the potential to revolutionize the oracle landscape in the blockchain industry and drive innovation forward. With its secure and efficient data delivery capabilities, TONSeer sets a new standard for oracle systems, empowering smart contracts and decentralized applications with accurate and reliable data.

References

[1] Roy Friedman, Achour Mostéfaoui, Michel Raynal, *Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Byzantine Systems.* Proceedings of the International Symposium on Reliable Distributed Systems, 228–237, 2004.

[2] P. Berman, J.A. Garay, *Cloture Votes: n/4-resilient Distributed Consensus in t + 1 rounds*, Mathematical Systems Theory, 26(1): 3–19, 1993, `https://doi.org/10.1007/BF01187072`.

[3] Péter Urbán, Naohiro Hayashibara, André Schiper, Takuya Katayama, *Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm*, Proceedings of the International Symposium on Reliable Distributed Systems, 4–17, 2004, `10.1109/RELDIS.2004.1352999`.

[4] C. Wintle, *Collective Wisdom: Methods of Confidence Interval Aggregation*, Journal of Business Research, 2014

[5] I. Yaniv, *Weighting and trimming: Heuristics for aggregating judgments under uncertainty*, Organizational Behavior and Human Decision Processes, 69(3), 237–249, 1997