Crisis: Probabilistically Self Organizing Total Order in Unstructured P·2·P Networks

⋆ Mirco Richter ⋆
(mirco.richter@mailbox.org)

July 14, 2019

1 Introduction

In their pioneering, but largely ignored work *"Byzantine-Resistant Total Ordering Algorithms"* [10], the authors Moser & Melliar-Smith established total order on network events utilizing a concept, best described as *virtual voting.* This simple yet ingenious insight achieves full asynchrony and close to optimal communication overhead as almost no additional information has to be send, besides the actual payload. Instead messages acknowledging other messages are just interpreted as virtual processes executing some consensus algorithm to decide the total order.

Unfortunately, Moser & Melliar-Smith's approach is based on a byzantine fault tolerant protocol, that inevitably depends on the number of participants, or the overall voting weight in the system. Those algorithms are therefore useless when it comes to fully local, unstructured Peer-to-Peer networks and their ever changing number of participants, or potentially unbounded voting power.

However, in *"Byzantine Agreement, Made Trivial"* [9], Micali described a so called player replaceable consensus protocol, that is able to execute each step of the computation inside an entirely different set of processes. *Player replaceability* is therefore a real paradigm shift when it comes to agreement in open systems. In particular, it allows for previously unknown solutions to the BFT-CUP problem as described by Alchieri et al. in [1].

It is true, that Micali's protocol needs some level of synchronism, but this is where the full power of virtual voting really shows up:

Inside the virtual setting, synchronism can be simulated, while keeping the actual system fully asynchronous from the outside. In fact such a behavior can be achieved, by simply interpreting messages as clock ticks of even length, regardless of the amount of real world time it took them to arrive. We call this phenomena *virtual synchrony* and point out, that it appears pretty straight forward from the internal logic of Lamport clocks.

With all this in mind, a combination of Micali's player replaceability and Moser & Melliar-Smith's idea of virtual voting seems quite obvious, provided the goal is total order on messages in unstructured Peer-to-Peer networks.

The Crisis protocol family follows this line of thought and presents a framework for asynchronous, signature free, fully local and probabilistically converging total order algorithms, that may survive in high entropy, unstructured Peer-to-Peer networks with near optimal communication efficiency. Regarding the natural boundaries of the CAP-theorem, Crisis chooses different compromises for consistency and availability, depending on the severity of the attack.

The family is parameterized by a few constants and external functions called voting-weight, incentivation & punishement, difficulty oracle and quorum-selector. These functions are necessary to fine tune the dynamics and very different long term behavior might appear, depending on any actual choice. Since proper function design is highly important, Cisis should be seen more as a total order framework, than as an actual algorithm.

Historical note

It is a concerning and somewhat wired incident in history that no reference to the foundational work of Moser & Melliar-Smith [10] seems to appear in any of the modern literature on virtual voting based approaches like hashgraph [2], parsec [5], or blockmania [6] at the time of this writing. In contrast, some of the later references make it look like virtual voting is a new invention. However it is neither new, nor is it an invention at all:

To the best of the authors knowledge, the ability to execute agreement protocols 'virtually' (by which we mean that "[..] *votes are not contained explicitly in the messages, but are deduced from the causal relationships between messages* [..]" [10, p.84]), was first observed by Moser & Melliar-Smith in 1993. The authors showed, that such a property is inherent in any so called byzantine partial order, which are more or less just cryptographically secured Lamport timestamps, that appear naturally whenever a message acknowledges another message. Virtual voting is therefore not a new invention, but an *emergent mathematical phenomena* on certain graphs, that have a structure similar to the one used in the Lamport clocks.

Remark. This strange glitch in time might exist, at least partially, due to monetary interest. It shines a scary light on the fragility of scientific standards, when huge econimic interest are suddenly involved. The antidote is proper education and any reader interested in historical correctness is encouraged to look at the origins of those ideas as started by Lamport [8] back in 1978.

After all, the work of Moser & Melliar-Smith [10] deserves all the credits, when it comes to virtual voting on byzantine partial order graphs. That paper has the potential to be seminal, despite the fact that it appears not referenced at all in most modern approaches, at the time of this writing.

License

Donations

If you would like to support the continuous production of content like this, please donate via one of the following channels, or contact the author for additional solutions:
Bitcoin:
1B5DNwRGC3Kb2MbPuJB4cQX9UsChPUvUWf
PayPal: mirco.richter@mailbox.org

2 Model of Computation

2.1 Random oracle model

We work in the random oracle model, e.g. we assume the existence of a cryptographic hash function, that behaves like a random oracle. We write

$$H : \{0,1\}^* \to \{0,1\}^p \tag{1}$$

for such a function, as it maps binary strings of arbitrary length onto binary strings of fixed length p. As usual, we call $H(b)$ the *digest* value of the binary string b and assume H to be collision-, preimage- and second preimage-resistant.

2.2 Network model

The protocol is executed in a dynamic, distributed system, where processes might join or leave at any time. We therefore have to assume some sort of simple message-oriented transport protocol, such that each participating processes is eventually able to send or receive data packages. In addition, the system is considered fully asynchronous in that no bound can be placed on the time required for a computation or for communication of any message.

A process is called *honest* at time t, if it executes the protocol according to the rules at that time and it is called *faulty* if it deviates from the protocol in one way or another.

3 Data Structures

3.1 Messages

Messages distribute payload across the network and the purpose of the present paper is to establishe a total order on those messages, that respects causality and is probabilistically invariant among all honest participants.

The system is open and anyone is able to inject an arbitrary amount of messages at any given moment in time. However we crucially require the existence of a function, that assigns a weight factor to any such message. The purpose of this weight is both to prevent Sybil and system scale DOS attacks and to provide any message with a certain amount of voting power to influence the generated order.

Proper weight function design is therefore of major importance when it comes to behavior control and self-organization. Different dynamics might appear relative to any weight function, some of which are stable and some of which are not.

To define our message type, we expand the ideas of Lamport [8] as well as Moser & Melliar-Smith [10] and use additional insight from blockmania [6] and Bitcoin [11]. Giving three fixed protocol constants $c1, c2, c3 \in \mathbb{N}$, a message is then nothing but a byte string of variable length, subject to the following interpretation:

```
struct Message{
    byte[c1] nonce,
    byte[c2] id,
    byte[c3] num_digests,
    byte[p * num_digests] digests,
    byte[ ] payload
}
```

In this definition, the *nonce* is a general purpose byte field of fixed length. It might be required to compute the weight function in an actual incarnation of the protocol. For example, if a protocol weight function is similar to Hashcashs Proof-of-Work [11], the nonce is necessary to probe the search space of hash values. If, on the other hand, the protocol uses a Proof-of-Stake or Proof-of-Authority style weight function, the nonce might contain a signature of the message to verify ownership of some staked voting weight.

In addition, payload can be anything, that is properly serializeable into a bytefield. Other then that, Crisis makes no assumptions on its internal structure. In any case, the outcome of the protocol is a total order, e.g. a chain of payload chunks.

Moreover, *id* is a binary string used to group messages into what we call virtual processes. Its neither a unique identifier of a message, nor must it represent an actual process. Its main purpose is to clearify how the ideas of BFT-CUP [1] emerge in our virtualized setting. The last messages of a virtual round with an identical *id* will be considered as votes from the virtual process. However, depending on the weight function, it might be possible that different real world processes collaborate under the same virtual process *id*.

The *num_digests* field is just a standard way to represent the length of the following byte array *digests*, the latter of which contains digest values that acknowledge the existence of other messages, or the empty string, in case the message does not acknowledge any other message[1]. We assume, that *digests* contains any digest only once, which implies that we work with graphs not with multi-graphs, later on.

The key insight here is, that a message that acknowlede other messages defines an inherent natural causality. To the best of the authors knowledge, this by now standard mechanism was derived in great detail by Lamport in his paper [8] from 1978 and we encourage the interested reader to look at the original source for further explanations.

In any case, if m and $\acute{m}$ are two messages, we write

$$m \to \acute{m}\,, \tag{2}$$

if and only m acknowledges $\acute{m}$, that is the digest $H(\acute{m})$ of $\acute{m}$ is contained in the field $m.digests$. We then say that m is a **direct effect** of $\acute{m}$, or that $\acute{m}$ is a **direct cause** of m and that both are in a direct causal relation[2].

[1] Acknowledgement of the empty string is straight forward and easily definable as the hash of the empty string $H(\{\})$.

[2] We chose this arrow convention to be more in line with the ideas of BFT-CUP [1]. The arrow can be interpreted as "has knowledge of".

In what follows, we write $\textsc{message}$ for the set of all messages and postulate a special non-message $\oslash \in \textsc{message}$[3]. Moreover we assume the existence of a string metric $d : \textsc{message} \times \textsc{message} \to \mathbb{R}$ like the Levenshtein distance, such that $(\textsc{message}, d)$ is a metric space and we are able to talk about the distance $d(m, \acute{m})$ between two messages.

3.1.1 Weight systems

The protocol assumes the existence of a so called weight system, which assigns a certain value to any given message and defines a way to combine the weight of different messages. It also provides a minimum threshold on message weight for the prevention of Sybil attacks. The choice of such a system is crucial and the overall dynamic of the system depend on it.

Definition 3.1 (Weight system)**.** *Let* $(\textsc{message}, d)$ *be the metric space of all messages and* $(\mathbb{W}, \leq)$ *a totally ordered set. Then the tuple* $(\mathbb{W}, w, \oplus, c_{min})$ *is a* ***weight system****, if* w *is a function*

$$w : \textsc{message} \to \mathbb{W} \tag{3}$$

that assigns a an element of $\mathbb{W}$ *to any message, called the* ***weight function****,* $c_{min} \in \mathbb{W}$ *is a constant, called the* ***weight threshold*** *and* $\oplus$ *is a function*

$$\oplus : \mathbb{W} \times \mathbb{W} \to \mathbb{W} \tag{4}$$

called the ***weight sum****, such that the following characteristic properties are satisfied:*
– Tamper proof: Let $m \in \textsc{message}$ *be a message, with weight* $w(m) \geq c_{min}$ *and let* $\acute{m} \neq m$ *be another message, close to* m *in the metric* d*. Then* $w(\acute{m}) < c_{min}$*, with high probability.*
– Uniqueness: If there are two messages m *and* $\acute{m}$ *with* $m \neq \acute{m}$*, then* $w(m) \neq w(\acute{m})$ *with high probability.*
– Summability: $(\mathbb{W}, \oplus)$ *is a totally ordered, abelian group.*

[3]In what follows, this message will indicate the inability of the system to agree on any actual message in a given voting period.

Remark. If $(\mathbb{W}, w, \oplus, c_{min})$ is a weight system, we sometimes write $\ominus x$ to indicate the inverse of an element x in the group $(\mathbb{W}, \oplus)$ and $x \ominus y$ for the sum with such an inverse. Moreover, if M a set of messages, we write

$$w(M) := \bigoplus\nolimits_{m \in M} w(m) \tag{5}$$

for the sum of the individual weights of all messages from M and call it the (overall) weight of M. In addition we use the convention $w(\emptyset) = 0$, where 0 is the neutral element in $\mathbb{W}$.

Given any message $m \in \textsc{message}$, the value $w(m)$ is interpreted as the amount of voting power, m holds to influence total order generation. The temper proof property assures, that processes can not change messages easily, without dropping their weight below a certain threshold. As explained by Beck in his 2002 paper [3] on Hashcash, such an approach ensures resistance against Sybil and certain DOS attacks without the need for any Signature scheme. It is famously utilized in the Nakamoto consensus family [11].

However in contrast to Nakamoto consensus, the present protocols are leaderless and voting is a collective process, where the overall voting weight is a combination of individual weights. The system therefore needs a way to actually execute this combination. This is reflected in the weight sum operation $\oplus$[4].

3.1.2 Causality

Messages may contain digests of other messages, which in turn contain digest of yet other messages and so on. This represents quite literally a partial order of causality: For a message m, to incorporate an acknowledgement of another message $\acute{m}$, message $\acute{m}$ must have existed before m, which implies that we can talk about the past and the future of any given massage. However a message might neither be in the past nor in the future of another message and those 'spacelike' messages are therefore not comparable. The purpose of a total order algorithm is then to extend the causal order into a total order, such that all messages become comparable.

[4]Weight systems might use ordinary addition or multiplication as their weight sum definition, however other ways to combine individual weights might be more realistic in certain setups.

To the best of the authors knowledge, this natural idea appeared for the first time in 1978 as part of Lamports seminal paper [8] under the term happens-before relation. Another frequently used term is 'spacetime' diagram, because the causal partial order between messages behaves very much like a spacetime diagram in special relativity. It is famously used in Lamport timestamps and was later adopted by Moser & Melliar-Smith, as foundation for what we might now call *virtual agreement* or virtual voting[5]. The following definition provides our incarnation of Lamports original ideas, adopted to our messages type:

Definition 3.2 (Causality)**.** *Let $m, \acute{m} \in$ MESSAGE be two messages. Then $\acute{m}$ is said to **happen before** m, if $m = \acute{m}$ or if there is a (possibly empty) sequence of messages $m_1, \cdots, m_k$, such that $m \to m_k \to \cdots \to m_1 \to \acute{m}$. In that case we write $\acute{m} \leq m$, call m an **effect** of $\acute{m}$ and $\acute{m}$ an **cause** of m and say that there is a **causality chain** from $\acute{m}$ to m.*

*Comparable messages are moreover called **timelike**, while incomparable messages are called **spacelike**. If messages $\acute{m}$ and m are timelike, $\acute{m}$ is said to be in the **past** of m and m is said to be in the **future** of $\acute{m}$, if $\acute{m} \leq m$.*

3.1.3 Vertices

To establish our total order, messages have to be extended by a small amount of local voting data, that is not transmitted to other processes. In fact, no votes are send through the network at all, but are deduced from the causal relation between messages. This is a key characteristic of virtual voting based systems, explicitly stated by Moser & Melliar-Smith in [10]. We call such an extension a *vertex*:

```
struct Vertex{
    Message m,
    Option<uint> round,
    Option<boolean> is_last ,
    Option<TotalOrderSet<uint>> svp  ,
    Option<(Message,Option<boolean>)>[ ] vote,
    Option<uint> total_position
}
```

[5]Much later, algorithms like hashgraph [2], parsec [5], or blockmania [6] adopted this in one way or another, unfortunately without any reference to the original ideas.

We write VERTEX for the set of all vertices and assume that any entry of option type is initialized with the default value, which we symbolize as $\perp$. Properties of messages are then easily extended to corresponding properties of vertices and we write:

$$\begin{aligned} & w(v) \leftarrow w(v.m), \\ & v.nonce \leftarrow v.m.nonce, \\ & v.id \leftarrow v.m.id, \\ & v.num\_digests \leftarrow v.m.num\_digests, \\ & v.digests \leftarrow v.m.digests, \\ & v.payload \leftarrow v.m.payload \end{aligned} \tag{6}$$

If v is a vertex, $v.m$ is called the *underlying* message of v. It is important to note, that equal messages might not result in equal vertices, as the appropriate vertices might have otherwise different entries. We therefore have to loosen the rigidity of equality a bit and use the following definition of equivalence instead.

Definition 3.3 (Equivalence of vertices)**.** *Let v and $\acute{v}$ be two vertices with equal underlying messages, i.e. $v.m = \acute{v}.m$. Then v and $\acute{v}$ are said to be **equivalent** and we write $v \equiv \acute{v}$.*

The causal relation (3.2) between messages can then be extended to a causal relation between vertices.

Definition 3.4 (Vertex causality)**.** *Let $v, \acute{v} \in$ VERTEX be two vertices. Then $\acute{v}$ is said to happen before v, iff $\acute{v}.m \leq v.m$. In that case we call v an effect of $\acute{v}$ and $\acute{v}$ an cause of v and say that there is a causality chain from $\acute{v}$ to v. Comparable vertices are moreover called **timelike**, while incomparable vertices are called **spacelike**. If vertices $\acute{v}$ and v are timelike, $\acute{v}$ is said to be in the **past** of v and v is said to be in the **future** of $\acute{v}$, if and only if $\acute{v} \leq v$.*

3.2 Lamport graphs

As partially ordered sets are more or less the same thing as directed acyclic graphs by the categorical DAG $\models$ POSET adjunction [12, sec 5.1], sets of causaly ordered vertices have a natural graph structure, which we call a Lamport graph. As implicitly understood by Moser & Melliar-Smith [10], those graphs are well suited for the generation of total order on network events.

Nevertheless, care must be taken when it comes to an actual set of vertices, as such a set might not be ordered at all, if it contains a vertex without all its acknowledging vertices. This motivates our definition of Lamport graphs as a vertex set, closed under the causality relation:

Definition 3.5 (Lamport Graph)**.** *Let $V \subset \textsc{Vertex}$ be a finite set of vertices, such that V contains all vertices $\acute{v}$ with $\acute{v} \leq v$ for all $v \in V$, but no two vertices in V are equivalent. Then the graph $G = (V, A)$ with $(v, \acute{v}) \in A$, if and only if $v \to \acute{v}$ is called a* ***Lamport graph****. Moreover, if v is a vertex in a Lamport graph G, the subgraph G_v of G that contains all causes of v is called the* ***past*** *of v.*

Two Lamport graphs are said to be ***equivalent****, if they are isomorphic as graphs and their vertex sets are equivalent, that is every vertex in one graph has an equivalent vertex in the other and vice versa.*

Lamport graphs are directed and acyclic for all practical purposes, because the inducing causality relation (3.4) between vertices is a partial order, with very high probability. The proof of the following proposition makes this precise.

Proposition 3.6. *Let G be a Lamport graph. Then, for all practical purposes, G is directed and acyclic.*

Proof. The proof is based on the assumption, that our hash function practically prevents causality loops, in other words, it is infeasible to generate vertices $v_1, \dots, v_k$, such that $v_1 \to v_2 \to \cdots \to v_k$, but $v_1 = v_k$ for some $k \geq 2$. Under this assumption, definition (3.4) provides a partial order on a vertex set V and the proposition follows from the categorical adjunction between posets and directed acyclic graphs.

To see the partial order on V in detail, first observe that reflexivity is immediate, since any vertex causally follows itself by definition (3.4). Transitivity is deduced from (3.4) in a similar fashion, as $v \leq \acute{v}$ and $\acute{v} \leq \tilde{v}$ implies the existence of causal chains $\acute{v} \to v_k \to \cdots \to v_1 \to v$ and $\tilde{v} \to w_j \to \cdots \to w_1 \to \acute{v}$, which combine into a causal chain from $\tilde{v}$ to v, hence $v \leq \tilde{v}$.

We proof antisymmetry by contradiction and assume $v \neq \acute{v}$, but $v \leq \acute{v}$ as well as $\acute{v} \leq v$. Then there are causal chains $\acute{v} \to v_k \to \cdots \to v_1 \to v$ and $v \to w_j \to \cdots \to w_1 \to \acute{v}$, which implies that there is a causal chain loop $v \to w_j \to \cdots \to v_1 \to v$. This however violates our assumption on the infeasibility of generating those loops. $\square$

Since the geometric structure of a Lamport graph is fully determined by its underlying set of messages, the past of equivalent vertices is the same in any graph. This key feature is crucial in the generation of an invariant total order and the following key theorem makes this precise.

Theorem 3.7 (Invariance of the past)**.** *Let $v \in G$ and $\acute{v} \in \acute{G}$ be two equivalent vertices in two Lamport graphs. Then the past of v in G is a Lamport graph, equivalent to the past of $\acute{v}$ in $\acute{G}$, for all practical purposes and G_v and $\acute{G}_{\acute{v}}$ have equal cardinality, i.e. $|G_v| = |\acute{G}_{\acute{v}}|$.*

Proof. Recall that the cardinality of a finite graph is equal to the number of its vertices. We start our proof with the simple observation, that the past of a vertex in a Lamport graph is a Lamport graph, since it trivially contains all elements from its past. It therefore remains to show, that the vertex sets from G_v and $\acute{G}_{\acute{v}}$ are equivalent and of equal size.

To see that, first observe that $v.digests$ and $\acute{v}.digests$ actually contain the same digests, as $v.m = \acute{v}.m$ follows from our definition of equivalence (3.3). Since G is a Lamport graph, it must contain a set of v's direct causes $S_v := \{x \in G \mid H(x.m) \in v.digests\}$ and since $\acute{G}$ is a Lamport graph too, it must also contain a set of $\acute{v}$'s direct causes $S_{\acute{v}} := \{y \in \acute{G} \mid H(y.m) \in \acute{v}.digests\}$. However, since H is a cryptographic hash function, we know that $x.m$ is equal to $y.m$ with very high probability for all $x \in S_v$ and $y \in S_{\acute{v}}$, due to the second preimage resistance of our hash function H. This implies that all vertices in S_v and $S_{\acute{v}}$ are equivalent, with very high probability. Moreover S_v and $S_{\acute{v}}$ are of equal size, since no Lamport graph contains equivalent vertices.

The same argument can then be applied to all pairwise equivalent vertices $x \in S_v$ and $\acute{x} \in \acute{S}_{\acute{v}}$ with $x \equiv \acute{x}$, which proofs the proposition by induction, since both G_v and $\acute{G}_{\acute{v}}$ are finite. $\square$

Remark. In our incarnation, Lamport graphs do not necessarily represent actual network communication. All they represent is causal order between messages. Crisis therefore allows for 'ghost processes', which just route & distribute data without ever generating messages themselves. Those processes are entirely transparent from the inside of any Lamport graph and are forbidden per definition in 'gossip-over-gossip' style adaptations of Lamports original ideas as used in [2], or [5]. We believe that our approach is more general and works better under sophisticated byzantine behavior in fully local and unstructured Peer-2-Peer networks.

4 Communication

Crisis is build on top of two simple push&pull gossip protocols, that are used for the distribution of messages and to keep local knowledge of neighbors up to date. Such gossip algorithms are well suited for the communication in unstructured Peer-2-Peer networks, as seen in real world applications like Bitcoin. However, a developer is free to choose any other approach, if necessary. All the system needs, is a way to distribute messages in a byzantine prone environment.

4.1 Message generation

All network communication starts with the generation of messages which are then distributed using the protocols delivery system. However messages must satisfy a certain structure to be redistributed by any honest process. This is an effective first measure against the easily detectable part of faulty behavior. Algorithm (2) shows how a honest process generates a valid message m, assuming that *nonce* is chosen in such a way, that $w(m) > c_{min}$:

Algorithm 1 Generate message

```
1: procedure MESSAGE(id, nonce:n, load:p, lamport_graph:G)
2:     Find a last vertex $v$ with $v.id = id$ in $G$
3:     Choose $\acute{S} \subset \{\acute{v}.m \mid \acute{v} \in G \wedge \acute{v} \notin G_v\}$, such that
       all elements of $\acute{S}$ have different $id$'s
4:     return [n, id, $|\acute{S} \cup \{v.m\}|$, $\{H(\acute{m}) \mid \acute{m} \in \acute{S} \cup \{v.m\}\}$, p]
5: end procedure
```

According to algorithm (2), a honest process generates a message by including a digest of the last message that it knows with the same *id*. A message is called a last message of a given id, if it is not in the past of any other message under the same id. In addition a set of digests from messages is incorporated, that are not in the past of the already included last message with the same *id*. This latter set is otherwise undetermined by the protocol and any choice is valid.

Remark. In an actual application, a honest process might just incorporate acknowledgements to a random subset of all messages that it received after the generation of the previous message m with the same *id*. Those messages can not be in the past of m, due to the definition of Lamport graphs and are therefore valid. However the same process might as well apply a more sophisticated strategy for the inclusion of messages, depending on the incentivation and punishment strategy of the system.

If a valid message is generated, the appropriate process should generate an new vertex and write it into its own Lamport graph for further distribution. The following section describes the proper way to handle this situation.

4.2 Lamport graph extension

After some process obtains a byte string that might be a message, it has to rule out all immediately observable faulty behavior and then check the integrity of that message against its own Lamport graph. If everything works out, the Lamport graph is extended with a vertex including the new message, if not, the message is deleted.

If the message is not already known, the procedure starts with a low level check against the basic structure of a massage, including bound checks and things like that. We abstract this as a boolean valued function BYTELEVEL_CORRECTNESS. After that, the process checks the weight of the message to see if it is above the minimum threshold bound c_{min}. To check the payload, we assume the existence of a boolean function PAYLOAD_CORRECTNESS that compares the payload against the system rules.

If all this works out properly, the process checks the entries in m's field of digests $m.digests$. All referenced messages must have exactly one corresponding vertex in the current Lamport graph and all of theses vertices must have different *id*'s. The process then looks for vertices with the same id as the one in the message. If there are some in the current Lamport graph, the process makes sure that one of these messages is referenced in $m.digests$.

If any of this does not work out, the message is considered faulty and is deleted. If on the other hand, everything is ok, the Lamport graph is extended with a new vertex that contains the message and new edges that points from the vertex to all vertices with messages referenced in $m.digests$. Algorithm (2) shows the details:

Algorithm 2 Message integrity

```
 1: procedure INTEGRITY(message:m, lamport_graph:G)
 2:     if BYTELEVEL_CORRECTNESS($m$) and
            $w(m) > c_{min}$ and
            PAYLOAD_CORRECTNESS($m.payload$) and
            there is no vertex $v \in G$, with $v.m = m$ and
            every $H \in m.digests$ references a vertex in $G$ and
            all referenced vertices have different $id$'s
 3:     then
 4:         if there is a vertex $v \in G$ with $v.id = m.id$ then
 5:             $v$ is referenced in $m.digests$
 6:             No referenced vertex is in the past of $v$
 7:             return true
 8:         end if
 9:     end if
10:     return false
11: end procedure
```

To be more precise, we call a graph $\acute{G}$ an extension of a Lamport graph G, by a vertex v, if and only if $\acute{G} - G = v$, e.g if G and $\acute{G}$ differ by v, only. As the following proposition shows, any extension of a Lamport graph, is itself a Lamport graph.

Proposition 4.1 (Lamport graph extensions)**.** *Suppose that G is a Lamport graph and m a byte string with* INTEGRITY$(m, G) = true$*. Then the extension $\acute{G}$ of G by a vertex v with $v.m = m$ is a Lamport graph.*

Proof. The MESSAGE_INTEGRITY function implies that there is no other vertex in $\acute{G}$ that is equivalent to v and that all direct causes $v \to \acute{v}$ of v are elements of G, hence of $\acute{G}$. □

Message integrity detects most faulty behavior. However there is a kind of fault, called a *mutation*, that can not be ruled out in this way, because it is not strictly local and therefore undetectable from the outside of any Lamport graph. Such a mutation occurs, if a set of messages, all with the same id, properly reference one and the same previous message with that *id* in a Lamport graph. The set then mutates the causal chain of messages with the same *id* and these errors are mapped into the Lamport graph. They are the reasons for byzantine agreement to appear in the first place.

No message integrity check can rule this out, as such a failure occures only relative to other messages and those messages might arrive at different processes during different times. Moreover since we assume no signature scheme, every process can generate mutations for any *id* that it knows[6].

Definition 4.2 (Mutation)**.** *Let G be a Lamport graph. Then two vertices v and $\acute{v}$ in G are called a mutation of a virtual process, if they have the same id and are spacelike, i.e neither $v \leq \acute{v}$ nor $\acute{v} \leq v$ holds.*

Remark. Mutations like this are called forks in the hashgraph consensus paper [2], which describes the situation quite nicely. However we stick to the original term as defined by the actual providers Moser & Melliar-Smith to properly honor their contribution as it should be.

The possibility of mutations is the true reason, why total order algorithms from the Moser & Melliar-Smith family need byzantine fault tolerance. Messages are considered as votes from virtual processes and mutations mimic byzantine behavior in actual voting systems, where an actor might deliver different votes to different processes. This is exactly the situation, that byzantine fault tolerant protocols deal with.

[6]However the system might be designed in such a way that certain *id*'s a economically favored over others, for example if a reward is associate to it that can only be accessed by the original creator of that *id*.

4.3 Member discovery gossip

We view the system as a dynamic, directed graph, where vertices are processes and an edge indicates the current ability of a process to send a message to another process. We follow [1] in their notation and write $\Pi(t)$ for this graph, as it would appear to an omnipotent outside observer. However no process must know the entire system and each $j \in \Pi(t)$ might have a partial view $\Pi_j(t)$ only. By definition, process j is then able to send data to any member k of its local view, but to no other participant. Our system therefore meets the criteria of a proper, unstructured Peer-to-Peer network.

Assuming a solution to the bootstrapping problem, every honest process j, knows a partial view, strictly larger then itself. The first gossip protocol is then a classic *process discovery gossip*.

It consists of a standard push&pull gossip, which means, that any honest process will choose another process periodically (but asynchronous, i.e. clock ticks are entirely local) at random and sends it a list of processes that it thinks are currently participating in the protocol and a list of processes, that it thinks have (temporally) left the system. In addition it will choose a random process, to ask it for a list of participating and leaving peers. In turn, if a honest process receives such a request, it sends a list of members that it thinks are currently participating in the network and a list of leaving processes in return. Algorithm (3) gives an example way to realize this protocol.

Algorithm 3 Process discovery

```
    run the following two loops in parallel forever

 1: loop discovery push&pull
 2:     wait for Poisson clock tick
 3:     send subset of $\Pi_j$ to random process $k \in \Pi_j$
 4:     send discovery requests to random process $k' \in \Pi_j$
 5: end loop

 6: loop
 7:     wait for data package
 8:     if data is a set of processes then
 9:         update $\Pi_j$
10:     else if data is process discovery requests then
11:         respond with subset of $\Pi_j$
12:     end if
13: end loop
```

As this goes on forever each process j will have an ever changing partial view $\Pi_j(t)$ into the system and it is free to restrict the amount of neighbors $|\Pi_j(t)|$ it knows, to not store to much data. No stop argument is involved and the network load must be regulated by the participating processes them self.

Depending on the actual churn, the frequency of the communication, that is the rate of the local clock ticks might be rather low, for example in the range of minutes. The purpose of this protocol is just to keep $\Pi_j(t)$ up to date, which enables a process to send and receive data from other processes.

Remark. A system engineer might incorporate additional stragegies to make communication etween honest processes more likely. However we leave this question open for further development.

4.4 Message gossip

Assuming that a process has a partial view $\Pi_j(t)$ into the network that is not completely wrong, it participates in the message gossip, which is the second asynchronous push&pull gossip. Its purpose is to distribute messages through the current population $\Pi(t)$.

Algorithm 4 Message gossip

```
    run the following loops in parallel forever

 1: loop send push&pull                              ▷ On many threads
 2:     wait for Poisson clock tick
 3:     send $S \subset \{v.m \mid v \in G \wedge v.total\_position = \bot\}$
        to random process $k \in \Pi_j$
 4:     send request for missing messages to random
        process $k' \in \Pi_j$
 5: end loop

 6: loop receive
 7:     wait for data package $m$
 8:     if MESSAGE_INTEGRITY$(m, G)$ then
 9:         expand $G$ with vertex $v$, such that $v.m = m$
10:     else if data is message requests then
11:         respond with appropriate set of messages
12:     end if
13: end loop
```

Messages are retransmitted via push gossip, only if they don't have a total order yet. This is the 'stop' criterion, required by high frequency push gossip protocols in general. Already ordered messages are pushed only as a response to a pull request.

However, despite such a stop criteria, message gossip never really stops, even if the production of new messages comes to a hold. This happens because without new messages, some of the previous ones might not achieve an order and are therefore retransmitted forever. For the system to be live we therefore have to make the assumption that new messages appear forever.

5 Total Order

Crisis extends the timelike causality between messages into a probabilistically converging total order, that enables comparison of spacelike messages in an invariant way.

Convergence happens as long as the network is able to estimate the overall amount of voting weight per time and the majority of processes behind that weight are interested in a stable order. We call this estimation a *difficulty oracle*, because it might behave very much like Bitcoins difficulty function in certain implementations. Fortunately, proper behavior is incentivizeable and deviation can be punished. The system therefore utilizes economical interest to achieve convergence.

Total order is then generated in four basic steps: The Lamport graph is divided into rounds and each round is tested for the occurrence of a so called safe voting pattern. Every time such a pattern appears, a next step in Micali's player replaceable agreement protocol BA^* is executed to locally decide a *virtual round leader* vertex.

Under partitioning, this leader might not be unique and a selection process similar to Bitcoins longest chain rule is applied such that all Lamport graphs eventually converge on the same round leader. In any case, the past of these round leader vertices is ordered concurrently to the rest of the system, using some kind of topological sorting, like Kahn's algorithm in combination with the voting weight to decide spacelike vertices. As the virtual round leader converges to a fixed value, so does the order.

5.1 Votes

As pioneered by Moser & Melliar-Smith [10], total order is achieved, if vertices vote on other vertices in some kind of virtual byzantine agreement process. Therefore, each vertex v has a field $v.vote$, where the entry $v.vote(r) = (l, b)$ describes v's vote (l, b) on some message $l \in \textsc{message}$, together with a possibly undecided binary value $b \in \{\perp, 0, 1\}$ in a so called round r.

5.2 Virtual communication

To appreciate the idea of virtual voting and to see how algorithm (7) works, we need to understand the information flow between so called *virtual processes* inside any given Lamport graph. In fact some virtual process id is able to pull information from another virtual process id', if and only if there are appropriate vertices v and $\acute{v}$, such that $v.id = id$, $\acute{v}.id = id'$ and $\acute{v}$ is in the past of v. In that case, we say that virtual process id received votes $\acute{v}.vote$ from virtual process id' and that there is a communication channel from v to $\acute{v}$.

Simulated communication channels like this are of course tamper proof and invariant among all Lamport graphs, due to the invariance of the past theorem (3.7). However byzantine behavior might still appear in the form of mutations and strategic, non random, message distribution.

Strategic message dissemination occurs, because any real world process is able to deviate from random gossip and send certain messages to certain peers only. In addition carefully mutated vertices might show different votes from the same virtual process, because any message creator is relatively free in choosing the past of any message. The overall effect is a virtual voting equivalent to the well known phenomena of byzantine actors sending different votes to different processes. A situation well suited for byzantine agreement protocols.

Although we have to accept such a behavior to some degree, it is nevertheless possible to prohibit strategic message distribution from any bounded adversary, who is able to manipulate an overall amount of voting weight k only.

One way to achieve this is by sending virtual votes through vertex disjoint path, only if the combined weight of their leightest vertices is greater then k. Such a strategy would be a weighted virtual interpretation of the message dissemination algorithm from [1].

However counting disjoint paths is computationally expensive and not really necessary in our setting, because virtual communication channels are already tamper proof from the inside. All we need is some insurance, that information flows through enough vertices from different real world processes. Such a requirement makes proper message distribution likely and counteracts any partition tendency to some degree. In a addition, it can be measured efficiently by just counting the overall weight in all path between two vertices.

Definition 5.1 (k-reachability)**.** *Let k be a positive number, G a Lamport graph and $v, \acute{v} \in G$ two vertices. Then $\acute{v}$ is said to be k-reachable from v, if the overall weight of all vertices in all path from v to $\acute{v}$ is greater then k. In that case we write $\acute{v} \leq_k v$.*

If we interpret a Lamport graph such that a *byzantine resistant* virtual communication channel exists from $\acute{v}$ to v only if $\acute{v} \leq_k v$, we ensure that k-bounded collaborations can not influence virtual communication channels by strategic message distribution.

5.3 Virtual synchronism

Lamport graphs represent a timelike order between vertices, that we interpret as virtual communication channels. Going one step further, we can forget about the outside world altogether and just think from the inside of a Lamport graph to define a virtual clock tick as a transition from one vertex to another.

This simple idea allows for internal synchronism, that enables us to execute strongly synchronous agreement protocols like Feldman & Micali's algorithm BA^* [9] virtually, but without any compromise in external asynchronism.

Remark. Again this insight was already present in the work of Moser & Melliar-Smith [10] under the term 'stage'. In fact it appears quite natural from the perspective of the well known Lamport clocks.

Any byzantine resistant protocol is based on the assumption that the amount of faulty behavior does not exceed a certain fraction of the overall voting weight and Crisis utilizes such a threshold too. However in contrast to most approaches, consistency does not depend on it, e.g. the order does not fork, even if the bound gets broken from time to time. Crisis therefore favors consistency over availability in such a scenario.

In any case, we need a way to approximate the voting weight, that is generated in a round and we must assume, that from time to time, not more then 1/3 of this weight is faulty.

We call such an approximation a *difficulty oracle*, because it might behave very much like Bitcoins difficulty function in certain Proof-Of-Work based incarnations. In any case, it is considered as an external parameter and different choices might lead to different behavior.

Definition 5.2 (Difficulty oracle)**.** *Suppose that the tuple $(\mathbb{W}, w, c_{min}, \oplus)$ is a weight system of the protocol. Then the function*

$$d : \mathbb{N} \to \mathbb{W} \tag{7}$$

that maps natural numbers onto weights, is called a ***difficulty oracle*** *w.r.t. the weight system and the value $d_r := d(r)$ is called the* ***round*** *r* ***difficulty*** *of the system.*

Example 1. The most simple example would be to just use a fixed constant that does not change over time as the systems difficulty oracle. This however might be way to simple for certain choices of weight systems, as we know from protocols like Bitcoin, that the overall voting weight per time (hash power for that matter) might fluctuate considerably. The overall goal is to compute a difficulty oracle such that equation (8) holds approximately.

Example 2. A more flexible difficulty oracle would hardcode its value for the first few rounds and then base the computation on the overall voting weight in the past of converged virtual round leader vertices later on. That past is invariant and therefore every process would compute exactly the same function.

With such a difficulty oracle at hand, we can look at algorithm BA^* as explained in [9], to see that it executes a potentially unbounded amount of synchronous rounds, each of which starts with a communication step, where any actor receives votes broadcast by the actors of the previous round. Our goal in this section is therefore to simulate that behavior, using the idea of internal time in combination with our byzantine safe communication channels (5.1). Algorithm (5) gives the details.

Algorithm 5 Virtual synchronous rounds

```
Require:
    connectivity $k$
    difficulty oracle $d$

procedure ROUND(vertex:v, lamport_graph:G)
    $N_v \leftarrow \{\acute{v} \in G \mid v \rightarrow \acute{v}\}$
    $r \leftarrow max(\{\acute{v}.round \mid \acute{v} \in N_v\} \cup \{0\})$
    if there is a $\acute{v} \in N_v$ with $\acute{v}.is\_last$ and $\acute{v}.round = r$ then
        $v.round \leftarrow r + 1$
    else
        $v.round \leftarrow r$
    end if
    $S_v \leftarrow \{\acute{v} \in G \mid \acute{v}.round = v.round - 1, \acute{v}.is\_last, \acute{v} \leq_k v\}$
    if $w(S_v) > 3 \cdot d_r$ then
        $v.last \leftarrow true$
    else
        $v.last \leftarrow (r = 0)$
    end if
end procedure
```

The procedure assumes a previous execution on all vertices in the past of v, but it can be called concurrently on spacelike vertices.

The algorithm computes so called *round numbers* and the *is_last* property of any vertex. The round number of a vertex is computed by first taking the largest round of all direct causes as its current estimation. If the vertex is a direct effect of a current round vertex with the *is_last* property, a new round begins and the vertex is a first vertex in that new round. If the vertex has enough last vertices of the previous round in its past and it is k-reachable from all of them, the vertex becomes a last vertex in its own round.

Last vertices are interpreted, as sending and receiving votes through byzantine resistant virtual communication channels to and from last vertices of consecutive rounds. This way, last vertices model the behavior of actors sending votes to other actors, whenever a round transition happens in algorithm BA^*.

Remark. The appearance of new rounds can not be guaranteed, even if we assume new messages to arrive forever. This is because the required high interconnectivity between messages must not happen. Extreme situations are thinkable, where no message references any other message and the Lamport graph is totally disconnected. Then, of course, no interconnectivity occurs and all message have a round number of zero forever. Any actual incarnation therefore requires proper incentivation to encourage the appearance of new rounds. This is possible, for example, if only vertices with $v.is\_last = true$ are incentivised in one way or another by the systems incentivation function.

Now, to understand our concept of virtual rounds a bit better, we proof a series of statements, that basically show that the round number and the *is_last* property are well defined and behave as expected. We start by showing that both properties do not depend on the actual Lamport graph, but are the same for equivalent vertices.

Proposition 5.3 (Round invariance)**.** *Let v and $\acute{v}$ be two equivalent vertices in Lamport graphs G and $\acute{G}$ respectively. Then $v.round = \acute{v}.round$ and $v.is\_last = \acute{v}.is\_last$.*

Proof. Both, the round number and the *is_last* property depend on certain sets of vertices in the past of a vertex, only. But since v and $\acute{v}$ are equivalent, they have equivalent pasts G_v and $\acute{G}_{\acute{v}}$, due to the invariance of the past theorem (3.7). We can therefore proof the statement by strong induction on the number of vertices, both in G_v and $\acute{G}_{\acute{v}}$.

For the base case assume that G_v contains v only. In that case $\acute{G}_{\acute{v}}$ contains $\acute{v}$ only and both N_v and $N_{\acute{v}}$, are empty. Then $v.round = 0$ and $\acute{v}.round = 0$, since algorithm (5) executes line (7) in both cases. Moreover, S_v and $S_{\acute{v}}$ are empty which implies $v.is\_last = true$ and $\acute{v}.is\_last = true$, since algorithm (5) executes line (13) in both cases and $r = 0$.

For the induction step, assume that G_v and $\acute{G}_{\acute{v}}$ are given and that $x.round = \acute{x}.round$ as well as $x.is\_last = \acute{x}.is\_last$ holds for all equivalent verices x and $\acute{x}$ in all Lamport graphs, with $|G_x| < |G_v|$ as well as $|G_{\acute{x}}| < |G_v|$.

Then there is exactly one $\acute{x} \in N_{\acute{v}}$ for every $x \in N_v$ and $x.round = \acute{x}.round$ as well as $x.is\_last = \acute{x}.is\_last$, since N_v and $N_{\acute{v}}$ are equivalent by the invariance of the past theorem (3.7) and $|G_x| < |G_v|$ as well as $|G_{\acute{x}}| < |G_v|$. This however implies $v.round = \acute{v}.round$, because algorithm (5) computes the same value r both for v and $\acute{v}$ and decides the same branch in line (4).

A similar reasoning shows $x.round = \acute{x}.round$ as well as $x.is\_last = \acute{x}.is\_last$ for all $x \in S_v$ and $\acute{x} \in S_{\acute{v}}$ and that $w(S_v) = w(S_{\acute{v}})$ holds, since equivalent vertices have equal weight. Hence both executions of algorithm (5) decide the same branch in line (10) and the proposition holds on G_v and $\acute{G}_{\acute{v}}$, which proof the proposition in any case by strong induction. □

Round numbers are compatible with causality, in the sense that the round number of a future vertex is never smaller then the round number of any vertex in its past. Round numbers are therefore an important first step in any attempt to totally order a Lamport graph. The following proposition gives the details.

Proposition 5.4. *Let v and $\acute{v}$ be two vertices in a Lamport graph G, such that $\acute{v} \le v$ holds. Then $\acute{v}.round \le v.round$.*

Proof. To see this, first assume $v \to \acute{v}$. Then $\acute{v} \in N_v$ and algorithm (5) computes $v.round \ge \acute{v}.round$ and the statement holds. The general situation then follows by repeated execution of (5) on each vertex in the causal chain $v = v_1 \to v_2 \; \cdots v_{n-1} \to v_n = \acute{v}$. □

A vertex has the *is_last* property, if and only if it is indeed a last vertex in a given round, i.e. every vertex in its future has a higher round number. We can therefore interpret these vertices as the end of a step in virtual BA^* and as the exact point in internal time, where a virtual process sends its vote to members of the next step. This serves as the basis for our virtual adaptation of algorithm BA^*.

Proposition 5.5 (Last vertices of a round)**.** *Let v be a vertex in a Lamport graph, with $v.is\_last = true$. Then every vertex in the future of v has a round number, strictly larger then $v.round$.*

Proof. Let $\acute{v}$ be a vertex in the future of v. Then, there is a path $\acute{v} \to v_1 \to \cdots \to v_k \to v$ in any Lamport graph, that contains $\acute{v}$ (and therefore v) and $\acute{v}.round \ge v_k.round$ follows from proposition (5.4).

However, since $v \in N_{v_k}$ and $v.is\_last = true$, either $max\{\tilde{v}.round \mid \tilde{v} \in N_v\} > v.round$, or algorithm (5) executes line (5). In any case, the round number of v_k is strictly larger then the round number of v and we get $\acute{v}.round \ge v_k.round > v.round$. □

If we consider vertices of a given round to receive votes from vertices of a previous round, we have to be sure, that those previous round vertices are indeed in the past of any current round vertex. The following proposition shows that this is indeed the case.

Proposition 5.6. *Let G be a Lamport graph and v a vertex with a positive round number $v.round > 0$ in G. Then v has at least one last round s vertex in its past for all round numbers $s < r$.*

Proof. We show the proposition for $s = r - 1$. The general case then follows by recursion, since G is finite.

To see the statement, observe that for a vertex to be in round r, r must either be the largest round number of its direct causes, or it must have a direct cause of round number $r - 1$ that is a last vertex of that round.

The second case is immediate. For the first case the argument can be repeated with any round r direct cause. Since the graph is directed, acyclic and finite and any sink vertex has round 0, there must eventually be a round r vertex, that has no round r direct causes. □

5.4 Difficulty bounds

Both the voting weight and the number of virtual processes is potentially unbounded in any given round. It is true that we can approximately limit the amount of faulty behavior at every moment by proper incentivation and punishment, but given enough time, byzantine behavior accumulates in the graph. In fact everyone can add arbitrary amounts of vertices with arbitrary large weights into any round, provided that sufficiently many new rounds appeared ever since.

For example a system with a Bitcoin-style Proof-Of-Work voting weight, might observe the occasional occurrence of something like a hash-bomb, i.e. a super heavy message that suddenly appears, but references messages way back in the past only. Such a 'bomb' is able to break all global byzantine bound assumptions in any round and it certainly exists if some motivated process puts all its hashing power for weeks, or even years into the generation of just one single message.

Another extreme example would be some kind of Internet-meme like phenomena, where suddenly large amounts of small to medium size messages occur in very old rounds for no apparent reason. In particular anybody can generate new messages in round zero easily, by not referencing other messages at all.

We might call fringe cases like this *time travel attacks*. The underlying reason is, that byzantine behavior is unbounded altogether, despite the fact that we can assume it to be approximately bounded at any given moment in time by our difficulty oracle.

According to Brewers CAP-theorem, behavior like this is unavoidable in any open and asynchronous system, because partition happens in unstructured systems without any governance, stake-, or member-lists in one way or another. It sharply distinguishes our situation from more traditional approaches like hashgraph [2], parsec [5], or blockmania [6] and puts our algorithm much more closely to Nakamoto's consensus.

Thats being said, unbounded byzantine behavior never happens in the past of any vertex, because that past is fixed forever, due to the invariance of the past theorem (3.7). We can therefore counteract such an attack locally, by carefully computing all relevant properties relative to the perspective of a vertex in a consecutive round only. The price to pay is globality, because agreement is achieved locally only.

Now, ideally, that is in an imaginative system without partitions, a system engineer would design the difficulty oracle such that the overall voting weight w_r^G of last messages in round r of Lamport graph G would always be in the range $3 \cdot d_r < w_r^G \leq 6 \cdot d_r$. This would guarantee any local round leader to be the global round leader and the order would be strictly convergent, not just probabilistically.

However, time travel attacks, forking and partitions are something to consider and because of that, the overall voting weight of a round is undefined, must not converge and varies between different Lamport graphs. The difficulty oracle can therefore be designed in such a way that an overall weight $> 3 \cdot d_r$ eventually happens frequently, but an upper bound estimation is impossible in general.

On the other hand, it is still rational to assume that the voting weight *per time* is in a certain range, at least approximately. Fortunately, this is enough to compute a theoretical upper bound on the overall amount of voting weight that might occur in any Lamport graph G. This bound can then be used to guarantee probabilistic convergence of the total order.

To see that, let t be an external time parameter and $\Pi(t)$ the system at time t. Then the maximal round number r_t at time t is the maximum of all round numbers in all Lamport graphs of the system $\Pi(t)$ as it would appear to an omnipotent external observer[7].

We can use this number to give an upper bound on the amount of voting weight that occurs in the system. If $G_{max}(t)$ is the largest Lamport graph that exists in the system at time t and if w_s^G is the overall voting weight of all last vertices in some Lamport graph G that has a round number s, then we assume our difficulty oracle to be designed such that

$$\lim_{|G| \to |G_{max}(t)|} \sum_{s=0}^{r_t} \frac{w_s^G}{d_s} \leq 6 \qquad (8)$$

holds *approximately* for all external time parameters t. Of course this number is theoretical, as no actual process can compute it, because no process knows r_t, or $G_{max}(t)$.

Basically, this inequality expresses the idea that the difficulty oracle is designed such that the amount of voting weight per time, is limited and no more then $6d_s$ weight can be produced in any round on average. However, it is flexible enough to allow every process to append generated voting weight into any round that currently exists.

[7]Of course this number is entirely theoretical as no participant can actually know it.

5.5 Virtual process sortition

In [1], Alchieri et al. looked at byzantine agreement in systems with unknown participants (BFT-CUP) and gave sufficient conditions to solve it. Their reasoning is solid, but they didn't consider player replaceable protocols.

However with player replaceability in mind, the situation changes, because every step in the protocol is executable in an entirely different set of processes. This implies that new solutions might appear and indeed a family of such solutions was found by Chen & Micali in Algorand [4], where consensus in open systems becomes more or less a problem of synchrony and quorum selection. The latter of which can be nicely solved by cryptographic sortition.

Nevertheless, the present situation is somewhat orthogonal to Algorand, as we can simulate synchronism easily, but cryptographic sortition might not work in our virtual setup. We therefore face the problem of how to decide, which virtual processes should execute a step in the protocol. Moreover, as our system is open and asynchronous, an unbounded amount of virtual processes might appear in any round.

Fortunately we can put things into perspective and consider the past of a vertex only, which fixes the problem of unbounded vertices, relatively speaking. However there might still be too much entropy in the system and we need a way to deterministically compute a subset of virtual processes that is somewhat favorable in the execution of a next step in the agreement protocol.

We call such a mechanism a *quorum selector* and consider it as another important parameter in any actual incarnation. Like the voting weight, different quorum selector functions might lead to very different long term behavior and the author believes that it is currently impossible to decide which one performs best under any given circumstances.

In any case, quorum selector functions decide virtual processes, not vertices. We therefore need a way to go from vertices to virtual processes first. This however is efficiently done, by deriving another graph from any Lamport graph, that projects vertices of equal *id*'s together. The following two definitions make the idea precise.

Definition 5.7 (Relative subgraph of a round)**.** *Let s and r be two round numbers with $s < r$, G a Lamport graph, v a round r vertex in G and V_v^s the set of all round s vertices in the past of v. Then the subgraph $G_v^s := (V_v^s, A_v^s)$ of G, with $(x,y) \in A_v^s$, if and only if $x, y \in V_v^s$ and $x \to y$, is called v's* ***round*** *s* ***past*** *in G.*

Now, the transition from vertices to virtual processes is done, by collapsing all vertices with the same id into some kind of new meta-vertex in the so called quotient graph. The latter of which is nothing but a quotient object in the category of graphs.

Definition 5.8 (Knowledge graph)**.** *Let s and r be two round numbers with $s < r$, G a Lamport graph, v a last message in round r and G_v^s the round s past of v in G. Then the quotient graph $\Pi_v^s := G_v^s \backslash \simeq_{id}$ defined by the equivalence relation $x \simeq_{id} y$, if and only if $x, y \in G_v^s$ and $x.id = y.id$, is called v's* ***round*** *s* ***knowledge graph****.*

We write id for an equivalence class vertex $\{\acute{v} \in G_v^s \mid \acute{v}.id = id\} \in \Pi_v^s$ and call it a round s ***virtual process****, from the perspective of v.*

Given any Lamport graph, our definition of knowledge graphs is efficiently computable and can be stored with little additional overhead. It is directed, but in general not acyclic anymore and it resembles a virtual version of the knowledge connectivity graph from [1].

To understand the meaning of this graph, consider that a virtual process $id \in \Pi_v^s$ has a directed edge to another virtual process $id' \in \Pi_v^s$, if and only if there is a vertex $\tilde{v}$ with $\tilde{v}.id = id$ and a vertex $\acute{v}$ with $\acute{v}.id = id'$ in G_v^s, such that $\tilde{v} \to \acute{v}$. Hence any edge represents the knowledge a virtual process has about the existence of another virtual process relative to a given round.

The following two propositions show that knowledge graphs are indeed well defined and invariant among different Lamport graphs.

Proposition 5.9 (Existence)**.** *Let s and r be two round numbers with $s < r$, G a Lamport graph and v a last message in round r. Then the round s knowledge graph Π_v^s is well defined, directed and not empty.*

Proof. Since v is a last vertex in a round $r > s$, v must have round s vertices in its past due to proposition (5.6). This however implies, that G_v^s is not empty as a directed graph. In addition $\simeq_{id}$ is an equivalence relation on the vertex set of G_v^s, which implies that the quotient is a well defined, directed and not empty, by the general properties of quotient objects in the category of graphs. □

We call two knowledge graphs Π_v^s and $\Pi_{\acute{v}}^s$ equivalent, if their reference vertices v and $\acute{v}$ are equivalent. As the following proposition shows, equivalent knowledge graphs are isomorphic and their elements consist of equivalent vertices only.

Proposition 5.10 (Invariance of knowledge graphs). *Let s and r be two round numbers with $s < r$ and v as well as $\acute{v}$ two equivalent round r vertices in Lamport graphs G and $\acute{G}$, respectively. Then the knowledge graph Π_v^s of v is isomorphic to the knowledge graph $\acute{\Pi}_{\acute{v}}^s$ and the elements in each equivalence class $id \in \Pi_v^s$ are in one-to-one correspondence with equivalent elements in $id' \in \acute{\Pi}_{\acute{v}}^s$.*

Proof. The invariance of the past theorem (3.7) implies, that G_v^s and $G_{\acute{v}}^s$ are isomorphic and vertices with equal id's are in one-to-one correspondence. Hence their quotients under the $\simeq_{id}$ relation, are isomorphic. Moreover, each equivalence class $id \in \Pi_v^s$ consist of vertices from G_v^s that have the same id. However due to invariance of the past, these are in one-to-one correspondence with vertices in $\acute{G}_{\acute{v}}$ that project onto the appropriate id in $\acute{\Pi}_{\acute{v}}^s$. □

Now, given any knowledge graph, a quorum selector is nothing but a way to chose a subset of virtual processes from that graph. The members are then interpreted as to send and receive votes through their last vertices.

Definition 5.11 (Quorum selector). *Let s and r be two round numbers with $s < r$, v a last round r vertex in a Lamport graph G and Π_v^s the round s knowledge graph of v. Then a **quorum selector** QUORUM deterministically chooses a subset $Q_v^s \subset \Pi_v^s$, called v's round s quorum, such that Q_v^s and $Q_{\acute{v}}^s$ are equivalent for equivalent graphs Π_v^s and $\acute{\Pi}_{\acute{v}}^s$.*

Quorum selection serves as a kind of filter, to reduce the overall byzantine noise, that might appear in the voting process of fully open systems. Its purpose is to make the appearance of a so called *safe voting pattern* as defined in the next section, more likely.

Example 3 (Highest voting weight quorum). Voting weight of vertices can be combined into voting weight of appropriate equivalence classes in Π_v^s, if we define $w(id) := \bigoplus_{v \in id} w(v)$ for any $id \in \Pi_v^s$. This is invariant among equivalent knowledge graphs Π_v^s and $\Pi_{\acute{v}}^s$ and low weight mutations do not change that value much.

A quorum selector function is then given by first choosing the weakly connected component of Π_v^s, that has the highest combined voting weight and then by ordering all virtual processes in that component according to their individual weight. After that the quorum selector might takes the heaviest n vertices from it, where n is a suitable constant, that makes the appearance of enough last vertices with an overall voting weight strictly larger then $3d_s$ probable.

The reasoning here is, that by restricting to a weakly connected component, faulty behavior based on graph partition is reduced. Moreover different vertices will compute the same quorums, as it is unlikely that the voting weight will fluctuate that much, seen from the perspective of different vertices. Moreover, mutations will effect the votes of these sets the least, simply because the voting power of very heavy vertices is less affected by lightweight mutations.

5.6 Safe voting pattern

With a quorum selector function at hand, we can now look at the last vertices of all quorum members in a given round and see if they qualify as proper voting sets.

Similar to any other byzantine agreement protocol, our virtual leader election (7) is based on the assumption that the amount of faulty behavior is bounded and does not exceed a certain amount of the overall voting weight. If this holds true voting takes place, if not voting stalls until the situation eventually resolves.

The purpose of a *safe voting pattern* is therefore to make sure, that voting takes place in those rounds only, that have appropriately bounded byzantine behavior. As described in section (5.4), the overall amount of faulty behavior is necessarily unbounded in any round, as the system is open and fully asynchronous. However it is always bounded relative to the past of any vertex, simply because that past is frozen and does not change ever again, due to the invariance of the past theorem (3.7).

This leads naturally to our definition of safe voting patterns, but before we derive the details, we need to specify the concept of a voting set first.

Definition 5.12 (Voting sets)**.** *Let $k \in \mathbb{R}^+$ be a positive number, r and s two round numbers with $s < r$ and v a last round r vertex in a Lamport graph G. Then the set*

$$S_v(s,k) := \{x \mid x.id \in Q(v,s) \wedge x \leq_{(r-s)k} v \\ \wedge\ x.round = s \wedge x.is\_last = true\}$$

is called $v.id$'s round s ***voting set*** *and $v.id$ is said to receive voting weight from the members of $Q(v,s)$ through $S_v(s,k)$. In addition, if t is another round number, with $t < s$, $l \in$* MESSAGE *a message and $b \in \{\bot, 0, 1\}$ a possibly undecided binary value, then*

$$w(S_v(s,k), t, (l,b)) := \\ w(\{x \in S_v(s,k) \mid x.vote(t) = (l,b)\})$$

is called the overall voting weight for the round t vote (l,b) that $v.id$ receives from its voting set $S_v(s,k)$.

We moreover say that v receives a ***super majority*** *of voting weight for a round t vote (l,b) from its voting set, if $w(S_v(s,k),t,(l,b)) > w(S_v(s,k)) \ominus d_s$ and a* ***honest majority*** *of voting weight, if $w(S_v(s,k),t,(l,b)) > d_s$, where d_s is the difficulty oracle in round s.*

Voting sets are invariant among equivalent vertices in different Lamport graphs, due to the invariance of the past theorem and the same holds for voting weights w.r.t. any given vote. The following proposition proofs the first statement, however to proof the second one, we need to understand how voting weights are actually computed first. We will do this in the following section.

Proposition 5.13 (Voting set invariance)**.** *Let v and $\acute{v}$ be two equivalent vertices in Lamport graphs G and $\acute{G}$ respectively. Then the voting sets $S_v(s,k)$ and $S_{\acute{v}}(s,k)$ are equivalent, i.e. both sets are isomorphic and consists of equivalent vertices only.*

Proof. Since the quorum selector is assumed to be invariant w.r.t. to vertex equivalence, all defining properties are actually invariant, which in tuen implies the invariance of any voting set. □

Using our definition of voting sets, we are now able to compute a safe voting pattern in a round. Algorithm (6) gives the details and we assume that it is executed on any vertex after algorithm (5) only.

Algorithm 6 Safe voting pattern

```
Require:
    connectivity $k$
    difficulty oracle $d$

1: procedure SVP(vertex:v, lamport_graph:G)
2:     $v.svp \leftarrow \emptyset : \emptyset$                                   ▷ empty total order
3:     if $v.is\_last$ and
           there is a $\acute{s} < v.round$ with
             $3d_{\acute{s}} < w(S_v(\acute{s},k)) \leq 6d_{\acute{s}}$ and
             $x.svp = y.svp$ for all $x, y \in S_v(\acute{s},k)$ and
             $(x.svp \neq \emptyset$ or $s = 0)$ and
             $|w(S_x(t,k),u,(l,\bot)) \ominus w(S_y(t,k),u,(l,\bot))| \leq d_t$
             $|w(S_x(t,k),u,(\cdot,b)) \ominus w(S_y(t,k),u,(\cdot,b))| < d_t$
             $t \leftarrow max(x.svp)$ for $x \in S_v(\acute{s},k)$      ▷ $max(\emptyset) = -\infty$
             $\forall x, y \in S_v(\acute{s},k)$, rounds $u \in x.svp \backslash \{t\}$, votes $(l,b)$
       then
4:         $s \leftarrow$ maximum of all such $\acute{s}$
5:         $v.svp \leftarrow x.svp \cup \{s\} : s \leq s$ and $t < s$ for all $t \in x.svp$
6:     end if
7: end procedure

    The procedure assumes a previous execution on all vertices in
    the past of $v$, but it can be called concurrently on spacelike
    vertices.
```

Given any vertex v, algorithm (6) computes the totally ordered set $v.svp$, which is used to index round numbers that have safe voting patterns in the past of v. In particular, a voting set $S_v(s,k)$ is said to be a safe voting pattern, if s is the maximal round number, such that $S_v(s,k)$ has enough overall voting weight to execute a step in a byzantine agreement protocol, all members $x \in S_v(s,k)$ have equal total orders $x.svp$ and all safe voting patterns of all members do not differ too much in any of their votes on previous rounds.

In addition, algorithm (6) implies, that safe voting patterns are nested sequences, where the elements of one stage reference the elements of a previous stage and so on. The following proposition makes this precise.

Proposition 5.14. *Let v be a vertex with $v.svp \neq \emptyset$, $r = max(v.svp)$ and let $S_v(r,k)$ be v's safe voting pattern. Then $x.svp = v.svp \backslash \{r\}$ for all $x \in S_v(r,k)$.*

Proof. If $x \in S_v(r,k)$, then algorithm (6) computes the set $v.svp$ as $x.svp \cup \{r\}$. □

To properly speak about the distance between two safe voting patterns it is moreover advantageous to define a metric on any totally ordered set $v.svp$.

Definition 5.15 (Svp distance)**.** *Let v be a vertex with $v.svp \neq \emptyset$. Then the **svp distance** is the function*

$$d_{v.svp} : v.svp \times v.svp \to \mathbb{R} \tag{9}$$

where $d_{v.svp}(r,r) = 0$ and $d_{v.svp}(s,r)$ is otherwise defined for any $s, r \in v.svp$ with $s \neq r$ as the number of different elements between s and r in the internal order plus one.

Remark. Safe voting patterns are not guaranteed to exist in any round, for various reasons. One of which is that the voting weights might differ to much, due to too much mutations. It is therefore of importance for any system engineer to implement some way that makes safe voting pattern at least likely. Ideally exactly one safe voting pattern would appear in every round. The more the system deviates from this rule, the more rounds are needed to make progress in the total order generation.

On the bright side we know, that safe voting patterns are byzantine fault detectors, because they accurately measure the amount of mutations of quorum members. This is good news, as any such fault detector can then be used to implement some invariant way of incentivation and punishment, which in turn can be used to make safe voting patterns attractive and economically favorable. Moreover, the folk theorems of repeated games suggest that such a system can be guided into all kinds of behaviors.

5.7 Local leader election

Any safe voting pattern provides an environment for the execution of another step in a player replaceable byzantine agreement protocol. The algorithm we use is an adaptation of Chen, Feldman & Micali's protocol BA^*, to the setting of Moser & Melliar-Smith's idea of virtual voting on causality graphs in a BFT-CUP environment.

Loosely speaking, a local round leader is nothing but a message, that defines an invariant set of vertices in any Lamport graph, the latter of which is then integrated into the total order, using some kind of topological sorting. Leader messages are computed in a byzantine agreement process, because we need to be sure, that all Lamport graphs of honest processes agree on them, at least locally, i.e in the causality cone of a safe voting pattern.

In any case, execution of the agreement protocol start with an initial round leader proposal, computed by a so called INITIAL_VOTE function.

Definition 5.16 (Initial Vote)**.** *Let 2^{VERTEX} be the power set of our vertex type. Then an **initial vote** function is a map*

$$\text{INITIAL\_VOTE} : 2^{\text{VERTEX}} \to \text{MESSAGE} \tag{10}$$

that deterministically chooses a message from any given set of vertices, such that the outcome is the same for equivalent vertex sets.

Initial vote functions are a system parameter and different choices might lead to different long term behavior. Ideally, all members of a safe voting patter would always compute the same initial vote. In that case an actual virtual round leader $l \neq \oslash$ would be decided in just a few extra rounds. However due to mutations, different members might compute different initial votes. In that case, it is the task of the virtual leader election to agree on a message anyway.

Since it is almost never the case that all members of a safe voting pattern are in agreement on a leader right away, the next best thing is to have at least a super majority of voting weight for some message. Based on this insight, the following example might give a reasonable choice for an initial vote function, based on the voting weight of messages.

Example 4 (Highest weight). A simple yet fast implementation of the INITIAL_VOTE function is given by choosing the underlaying message of the highest voting weight vertex. Since we assume that it is infeasible to have different vertices of equal weight, such a choice is practically deterministic and the outcome depends on the underlying message only.

After initial votes are made, a byzantine agreement protocol is executed in a chain of safe voting pattern, that locally decides on a message. However as the system is open, asynchronous and the voting weight is eventually unbounded in any round, we can never rule out, that a different leader is decided for the same round in another partition of the system.

Because of that, algorithm (7) itself does not decide a global leader but adds any local decision to the set of all possible leader in a round. The result is a stream of candidate sets that we call the *global leader stream* of a Lamport graph.

Definition 5.17 (Leader Stream)**.** *Let G be a Lamport graph and $2^{(uint,\text{MESSAGE})}$ be the power set of indexed vertices. Then the function*

$$\text{LEADER}_G : \mathbb{N} \to Option\langle 2^{(uint,\text{MESSAGE})}\rangle$$

is called the ***global leader stream*** *of the Lamport graph, the set* $\text{LEADER}_G(r)$ *is called the* ***candidate set*** *for the virtual round r leader and some element* $(s,l) \in \text{LEADER}_G(r)$ *is a possible round r leader message l, locally decided in round s.*

Basically, algorithm (7) computes the votes of a vertex on every local leader election in previous rounds, based on the votes of all members in its safe voting pattern. The special character $\oslash$ is used to indicate, that no actual message could be decided in a round. If the vertex is able to locally decide a leader, the global leader stream is updated, using function (8) as a variation of Nakamoto's longest chain rule.

To be more precise, algorithm (7) computes v's votes in all currently active voting rounds, by looping through the elements of $v.svp$. Each such element indicates a round number and a different stage δ, the latter of which is measured by the position of that round number inside the total order of $v.svp$.

Algorithm 7 virtual leader elections

```
if $v.svp = \emptyset$ then
    $\text{LEADER}_G(r) \leftarrow \text{NAKAMOTO}(\text{LEADER}_G(r), \oslash, v.round)$
    return
end if
$s \leftarrow max(v.svp)$
$S \leftarrow v$'s safe voting pattern $S_v(s,k)$
$n \leftarrow w(S)$
for all $t \in v.svp$ do
    $\delta \leftarrow d_{v.svp}(s,t)$
    if $\delta = 0$ then                                   ▷ Initial leader proposal
        $v.vote(t) \leftarrow (\text{INITIAL\_VOTE}(S), \bot)$
    else
        $l \leftarrow$ message with highest round $t$ voting weight in $S$
        if $\delta = 1$ then                               ▷ Leader presorting
            if $w(S,t,(l,\bot)) > n - d_s$ then
                $v.vote(t) \leftarrow (l,\bot)$
            else
                $v.vote(t) \leftarrow (\oslash,\bot)$
            end if
        else if $\delta = 2$ then                          ▷ $BBA^*$ initialization
            if $l \neq \oslash$ and $w(S,t,(l,\bot)) > n - d_s$ then
                $v.vote(t) \leftarrow (l,0)$
            else if $l \neq \oslash$ and $w(S,t,(l,\bot)) > d_s$ then
                $v.vote(t) \leftarrow (l,1)$
            else
                $v.vote(t) \leftarrow (\oslash,1)$
            end if
        else
            if $\delta \mod 3 = 0$ then                    ▷ Coin fixed to 0
                if $w(S,t,(l,0)) > n - d_s$ then
                    $v.vote(t) \leftarrow (l,0)$
                    if $w(S,t,(l,0)) = n$ then
                        $\text{LONG\_CHAIN}(\text{LEADER}_G(t), l, s)$
                    end if
                else if $w(S,t,(l,1)) > n - d_s$ then
                    $v.vote(t) \leftarrow (l,1)$
                else
                    $v.vote(t) \leftarrow (l,0)$
                end if
            else if $\delta \mod 3 = 1$ then               ▷ Coin fixed to 1
                if $w(S,t,(l,1)) > n - d_s$ then
                    $v.vote(t) \leftarrow (\oslash,1)$
                    if $w(S,t,(l,0)) = n$ then
                        $\text{LONG\_CHAIN}(\text{LEADER}_G(t), \oslash, s)$
                    end if
                else if $w(S,t,(l,0)) > n - d_s$ then
                    $v.vote(t) \leftarrow (l,0)$
                else
                    $v.vote(t) \leftarrow (l,1)$
                end if
            else if $\delta \mod 3 = 2$ then               ▷ Genuine coin flip
                if $w(S,t,(l,0)) > n - d_s$ then
                    $v.vote(t) \leftarrow (l,0)$
                else if $w(S,t,(l,1)) > n - d_s$ then
                    $v.vote(t) \leftarrow (l,1)$
                else
                    $b_{coin} \leftarrow lsb(H(x.m))$ for max weight $x \in S$
                    $v.vote(t) \leftarrow (l, b_{coin})$
                end if
            end if
        end if
    end if
end for
```

Algorithm 8 Longest chain rule

```
1: procedure LONG_CHAIN(set⟨uint,MESSAGE⟩:S,MESSAGE:m,uint:s)
2:     if there is no $(t,l) \in S$ with $t > s$ then
3:         $S \leftarrow (S \backslash \{(t,l) \in S \mid t < s\}) \cup \{(s,m)\}$
4:     end if
5:     return $S$
6: end procedure
```

Any election starts with vertex v proposing its initial vote for a leader in v's own safe voting patter. This is the $\delta = 0$ stage of algorithm (7) and it mimics the initial vote assumption, made in the original BA^* algorithm.

After that, stages $\delta \in \{1, 2\}$, basically indicate the two execution steps in Feldman & Micali's gradecast algorithm GC, while all higher stages $\delta \geq 3$ indicate an execution step in Micali's binary agreement protocol BBA^*. Of course every such step is entirely virtual and no votes are actually send to other real world processes as explained previously in great detail.

The purpose of the $\delta = 1$ stage is to presort all initial votes the vertex received for some round leader message. In fact an actual message $l \neq \oslash$ can become a round leader only, if some vertex receives a super majority of voting weight for that message. If this does not happen, the outcome will be the non-leader $l = \oslash$. Therefore any initial voting weight function has to account for this to ensure liveness.

In stage $\delta = 2$ the output of gradecast is transformed into the input of BBA^*, to prepare for the local decision either on a single message l or the non-leader message $\oslash$. In this stage an actual leader $l \neq \oslash$ can be proposed only, if a honest majority of voting weight is received for that message.

For any stage with $\delta \geq 3$ and $\delta \bmod 3 = 0$ we are in a 'Coin fixed to zero' round, according to Micali's terms. If a vertex receives voting weight that is in agreement on a vote with zero binary part in such a round, it locally decides a leader and uses the longest chain rule (8) to update the global leader stream. Note however the absence of a stop criteria. This is necessary for the longest chain rule to work properly. We explain this in the next section.

Stage $\delta \geq 3$ and $\delta \bmod 3 = 0$ is analog, but the decision will always be the non leader $\oslash$ message.

For a stage with $\delta \geq 3$ and $\delta \bmod 3 = 2$ we are in a so called 'Genuine coin flip' stage and as usual, no decision is made in such a round. In BBA^* all peers broadcast a unique signature and the least significant bit of the smallest hash of those signature if interpreted as a float, is the same for all participant with probability 2/3, provided 2/3 of all peers are honest.

Our virtualization of such a 'common concrete coin' works as follows: Instead of sending unforgeable signatures, vertices virtually send their own hash and we choose the heaviest of theses hashes to take the least significant bit of it. These hash values are sufficiently unforgeable as the voting weight would drop below the c_{min} threshold if changed, by our tamper-proof assumption. Moreover, we can assume, that the bit b_{coin} is sufficiently random and the same with a non zero probability p_{coin} for all members of the safe voting pattern that contains v, because the amount of forking is limited in that voting set.

Remark. The reader should note, that no termination occurs in any local election. However, once a local leader is decided, every consecutive safe voting pattern, that has such a deciding vertex in its past, will decide the same value due to agreement stability (6.11). Hence any actual implementation can stop the local computation for that round and just update the next round accordingly. This is more efficient from an implementation perspective, but the author believes that writing the abstract algorithm without stop criteria is conceptually cleaner.

5.8 Total order

As time goes by and the Lamport graph grows, more and more round leaders are computed and incorporated into the global leader stream $\text{LEADER}_G(\cdot)$ using procedure (8). We call this function the *longest chain rule*, because it deletes all local leader messages decided previous rounds and keeps those computed in the maximum round number, only. It always chooses the longest chain, so to speak. As we will proof in section (6.4), this allows the set of each round r leader to eventually converge to a single element with probability one. Total order is then achieved by topological sorting on the past of appropriate vertices.

The intuition is that the local leader election on a round r never stops, as every new round $s > r$ that has a safe voting patter, recomputes the round r leader. This can be seen as a chain of rounds s_i, that all compute the round r leader $(l, s_1) < (l, s_2) < (l, s_3) < \ldots$, but anytime the overall voting weight of such a round exceeds the upper bound $6 \cdot d_{s_i}$, additional round leader might appear.

However, every time more then one round r leader appears in a Lamport graph, the chain forks, like $(l, s_1) < \{(l, s_2), (\acute{l}, s_2)\} < \{(l, s_3), (\acute{l}, s_3)\} < \ldots$. The longest chain rule then selects the maximum hight set of elements in this chain together with all forks that might occur in that set. The reason is, that forks will eventually decay away and a single chain with a single message will asymptotically remain, provided our estimation on the difficulty (8) holds.

Algorithm (9) then uses the stream $\textsc{leader}_G(\cdot)$ to compute the total order and as $\textsc{leader}_G(\cdot)$ converges, so does the order. It is executed in an infinite loop and in concurrence to the rest of the system.

Algorithm 9 Order loop

```
run the following loop forever

 1: loop update order
 2:     wait for $\textsc{leader}_G(\cdot)$ to change
 3:     $s \leftarrow$ min round of all changed $\textsc{leader}_G(\acute{s})$
 4:     $r \leftarrow$ max round of all $\textsc{leader}_G(\acute{r}) \neq \emptyset$
 5:     $v_{l_{s-1}} \leftarrow$ leader in highest round, smaller $s$ in $G$
 6:     for $s \leq t \leq r$ do
 7:         $n \leftarrow max\{v.total\_position \mid v \in Ord_G(v_{l_{t-1}})\}$
 8:         (randomly) choose $(p, l_t) \in \textsc{leader}_G(t)$
 9:         if $l_t \neq \oslash$ then
10:             $\textsc{order}(Ord_G(v_{l_t}), n)$          ▷ $v_{l_t}.m = l_t$
11:         end if
12:     end for
13: end loop
```

$Ord_G(v_l)$ past of leader vertex v_l without the past of all leader vertices in previous rounds.

Every time a round leader appears, or is updated, the algorithm executes a topological sorting algorithm on the past of all future leaders of the smallest updated leader, without reodering the past of previous unchanged leaders. Since that past is invariant between all Lamport graphs by the invariance of the past theorem (3.7), every process will eventually compute the same total order, provided the leader streams of all honest processes converges.

Moreover, since we use topological sorting, the generated order will be an extension of the partial causality between massages.

Efficient topological sorting is known, able to achieve logarithmic run time, if executed concurrently on spacelike vertices. However for the sake of simplicity we use Kahn's algorithm (10) as our example, to generate total order in linear runtime.

Algorithm 10 Total order using Kahn's algorithm

```
 1: procedure ORDER(dag:Ord(v), uint:last)
 2:     $n \leftarrow last + 1$
 3:     $S \leftarrow$ set of all elements of $Ord(v)$ with no outgoing edges
 4:     while $S \neq \emptyset$ do
 5:         remove $x$ with highest weight $w(x)$ from $S$
 6:         $x.total\_position \leftarrow n$
 7:         $n \leftarrow n + 1$
 8:         for each vertex $y \in Ord(v)$ with edge $e : y \rightarrow x$ do
 9:             remove edge $e$ from $Ord(v)$
10:             if $y$ has no other outgoing edge then
11:                 $S \leftarrow S \cup \{y\}$
12:             end if
13:         end for
14:     end while
15: end procedure
```

Kahn's algorithm in its arrow reversed incarnation, since we want to order the past before the future in any Lamport graph.

Since the weight is invariant among all equivalent vertices and it is practically impossible for two vertices to have the same weight, execution of (10) will give the same results in any Lamport graph, which establishes an invariant total order.

Remark. Of course sorting by voting weight is just an example. In fact any deterministic function able to decide elements from S in line (5) in an invariant way can be used.

5.9 The Crisis protocol

Finally, the overall algorithm works as follows: Member discovery (3) and message gossip (4) are executed in infinite loops, concurrently to the rest of the system. Ideally the message sending loop is executed on as many parallel threads as possible. This implies that an overall unbounded amount of new messages arrive over time due to our liveness assumption. In addition each processes may generate messages and write them into its own Lamport graph.

For each new set of messages that pass the integrity check, the Lamport graph is extended by an appropriate set of vertices V that contain those messages. We assume all elements of V to be spacelike and that all vertices in the past of V have already decided round numbers, safe voting patterns and votes. If this is not the case, V can easily be partitioned into sets of spacelike vertices and the protocol is executed on their past first.

Then, algorithms (5), (6) and (7) are executed in that order concurrently on each vertex from V. As these algorithms run, they will update the leader stream $\text{LEADER}_G(\cdot)$ in some way.

In addition, the total order loop (9) runs concurrently to the rest of the system and waits for updates of the leader stream. Depending on the actual order algorithm (10), additional threads might be required to execute exponentially fast topological ordering algorithms.

6 Correctness Proof

We show that the Crisis protocol family eventually converts a causal order on messages into a total order on vertices that is asymptotically identical at all nonfaulty processes in the system. In particular we adapt Moser & Melliar-Smith definition of total order [10] to our probabilist setting and proof that the following properties hold under the assumptions we make in section (6.1):

1. *Probabilistic Termination I.* The probability that a honest process j computes $v.total\_position = i$ for some position i and vertex v increases asymptotically to unity as the number of steps taken by j tends to infinity.

2. *Probabilistic Termination II.* For each message m broadcast by a non byzantine process j, the probability that a non byzantine process k places some vertex v with $v.m = m$ in the total order, increases asymptotically to unity as the number of steps taken by k tends to infinity.

3. *Partial Correctness.* The asymptotically convergent total orders determined by any two non byzantine processes are consistent; i.e., if any non byzantine process determines $v.total\_position = i$, then no honest process determines $\acute{v}.total\_position = i$, where $\acute{v} \not\equiv v$.

4. *Consistency.* The total order determined by any non byzantine process is consistent with the partial causality order; i.e. $\acute{v} \leq v$ implies $\acute{v}.total\_position \leq v.total\_position$.

6.1 Assumptions

Our byzantine fault resistant total order is based on the following list of assumptions.

1. *Random Oracle Model.* Cryptographic hash functions exist, are collision, first- and second-preimage resistant and behave like random oracles.

2. *Liveness.* At every moment in time, there are non-faulty processes that participate in the system and every such process must generate further messages that causally follow messages from other nonfaulty process.

3. *Message Dissemination.* If Lamport-graph G of process j contains a vertex v and Lamport-graph $\acute{G}$ of process k does not contain any vertex, equivalent to v and both j and k are honest and participate in the protocol, then there will eventually be a Lamport graph $\tilde{G}$ of process k, with $\acute{G} \subset \tilde{G}$ and $v \equiv \tilde{v}$ for some vertex $\tilde{v} \in \tilde{G}$.

4. *Existence of Weight Systems.* A weight system as defined in section (3.1.1) exists and allows for the definition of a difficulty oracle function $d : \mathbb{N} \to \mathbb{R}^+$, that satisfies (8) approximately.

5. *Quorum selector & safe voting pattern.* A quorum selector exists, such that safe voting pattern appear frequently, i.e. the probability p_r that round r has a safe voting pattern is non vanishing.

6. *Initial Vote.* The initial vote function is able to generate vertices with $l \neq \oslash$ in the presorting stage $\delta = 2$ of algorithm (7).

6.2 Invariance

Votes are well defined and equal for equivalent vertices among different Lamport graphs. This is the foundation of virtual voting, because any real world process knows, that any other process will compute the same votes with respect to equivalent vertices. In other words, votes are deducible from the causal relation between vertices and we must not send them.

Proposition 6.1 (Safe voting pattern invariance). *Let v and $\acute{v}$ be two equivalent vertices in Lamport graphs G and $\acute{G}$ respectively. Then $v.svp = \acute{v}.svp$ as well as $v.vote(t) = \acute{v}.vote(t)$ for all $t \in v.svp$.*

Proof. Both properties $v.svp$ as well as $v.vote$ depend deterministically on the past of v, only. However equivalent vertices have equivalent histories, due to the invariance of the past theorem (3.7). We therefore proof the statement by strong induction on the number of vertices $|G_v|$ in the histrory of v (which is equal to $|\acute{G}_{\acute{v}}|$).

For the base case assume that v and $\acute{v}$ are two equivalent vertices in Lamport graphs G and $\acute{G}$ respectively, such that G_v contains v only. Then $\acute{G}_{\acute{v}}$ must contain $\acute{v}$ only and $v.round = 0$ as well as $\acute{v}.round = 0$ follows. This however implies that no round numbers $s < v.round$ and $\acute{s} < \acute{v}.round$ exist and algorithm (6) computes the empty total order $v.svp = \emptyset : \emptyset$ as well as $\acute{v}.svp = \emptyset : \emptyset$ in both cases, since the 'if' branch after line (3) is not executed. After that, algorithm (7) executes line (3) both for v and $\acute{v}$ and we get $v.vote = \perp$ as well as $\acute{v}.vote = \perp$, as no safe voting pattern exist in the past of both v and $\acute{v}$.

For the strong induction step assume that v and $\acute{v}$ are two equivalent vertices in Lamport graphs G and $\acute{G}$ respectively and that $x.svp = \acute{x}.svp$ and $x.vote(t) = \acute{x}.vote(t)$ for all $t \in x.svp$ and equivalent vertices x and $\acute{x}$ in all Lamport graphs $\tilde{G}$ and $\hat{G}$ with $|\tilde{G}_x| < |G_v|$.

If $v.is\_last$ then $\acute{v}.is\_last$ and since the voting sets $S_v(s,k)$ and $S_{\acute{v}}(s,k)$ are equivalent for all $s < v.round = \acute{v}.round$, we know that their overall voting weight must be identical, i.e. $w(S_v(s,k)) = w(S_{\acute{v}}(s,k))$. In addition $x.svp = \acute{x}.svp$ as well as $x.vote(u) = \acute{x}.vote(u)$ holds for all $x \in S_v(s,k)$ as well as $\acute{x} \in S_{\acute{v}}(s,k)$ and $u \in x.svp$, by our induction hypothesis, since $|G_x| = |G_{\acute{x}}| < G_v$. This however implies that algorithm (6) executes the if branch in line (3) for v, if and only if it executes the same branch for $\acute{v}$. Therefore $v.svp = \acute{v}.svp$.

In case $v.svp = \emptyset$ and $\acute{v}.svp = \emptyset$, algorithm (7) computes $v.vote = \perp$ as well as $\acute{v}.vote = \perp$ and otherwise executes its for loop on the same round numbers t both for v and $\acute{v}$, using the same δ in both cases. However, the voting weights of $S_v(s,k)$ and $S_{\acute{v}}(s,k)$ are equal for any vote by our induction hypothesis, since $|G_x| = |\acute{G}_{\acute{x}}| < G_v$. Therefore algorithm (7) chooses the same branches both for v and $\acute{v}$, which implies $v.vote(t) = \acute{v}.vote(t)$ for all $t \in v.svp$, since INITIAL_VOTE is deterministic and gives the same result on equivalent voting sets and $lsb(H(v.m)) = lsb(H(\acute{v}.m))$.

Altogether we get $x.svp = \acute{x}.svp$ and $x.vote = \acute{x}.vote$ for all equivalent $x \in G_v$ and $\acute{x} \in \acute{G}_{\acute{v}}$, which proofs the proposition by strong induction. ☐

6.3 Virtual Leader Election

We proof that the virtual leader election algorithm (7) eventually decides a set of round leader with probability one. As algorithm (7) is an adaptation of Chen, Feldman & Micali's algorithm BA^*, we follow their ideas and divide the proof into two subproofs, the first of which shows the graded consensus properties and the second of which proof the binary consensus part.

Remark. In what follows we will frequently say that some vertex v executes the virtual leader election algorithm (7). This in agreement with our line of thought, because a vertex v represents a virtual process $v.id$. But of course algorithm (7) is executed by some real world process with input (v, G), where G is a Lamport graph that contains v.

Consensus protocols are based on a property called agreement, which basically means that all honest processes hold the same value. However when it comes to weighted consensus it might not be appropriate to distinguish between a honest and a faulty process, but to talk about honest and faulty weight instead. We therefore say that the honest voting weight is in agreement on some vote, if all but a possibly byzantine amount of weight agrees on that vote.

Definition 6.2 (Agreement)**.** *Let v be a vertex with $|v.svp| \geq 3$, $\{r > s\} \in v.svp$ the largest two elements and $S_v(r,k)$ the safe voting pattern of v. We then say that the members of $S_v(r,k)$ are in **agreement** on some round $t \in v.svp \backslash \{r,s\}$ vote (l,b), if the voting set $S_x(s,k)$ of each such member $x \in S_v(r,k)$ has a super majority of voting weight for (l,b), that is the inequality $w(S_x(s,k),t,(l,b)) > w(S_x(s,k)) \ominus d_s$ holds.*

As the following corollary shows, our definition of agreement, immediately implies, that all members of a voting set compute the same vote for any agreed on value.

Corollary 6.3. *Let v be a vertex with $|v.svp| \geq 3$ and $\{r > s\} \subset v.svp$ the largest two elements, such that the members of $S_v(r,k)$ are in agreement on some round $t \in v.svp \backslash \{r,s\}$ vote (l,b). Then $x.vote(t) = y.vote(t)$ for all $x,y \in S_v(r,k)$.*

Proof. Each member $x \in S_v(r,k)$ executes algorithm (7) to compute its own votes. Since $t < s < r$, we know $t < r-1$, which implies that the 'Initial leader proposal' branch $\delta = 0$ is never executed for any $x \in S_v(r,k)$. But since a super majority of voting weight from $S_x(s,k)$ votes for (l,b), line (13) computes the same l again and one of the following branches decides the same $\acute{b} \in \{\perp, 0, 1\}$ for all $x \in S_v(v,k)$. Hence $x.vote(t) = y.vote(t)$ for all $x,y \in S_v(r,k)$. □

6.3.1 Graded Consensus

We start the correctness proof with a series of propositions, that basically show that the first three branches (i.e. $\delta \in \{0,1,2\}$) of our virtual leader election (7) are nothing but an adaptation of Feldman & Micali's graded consensus, but executed in three consequitive safe voting patterns. Our proofs are strongly influenced by the approach taken in [7].

Proposition 6.4 (Initial super majority)**.** *Let v be a vertex with $|v.svp| \geq 3$ and $\{r > s > t\} \subset v.svp$ the three maximum elements from $v.svp$. If there is a member $x \in S_v(r,k)$ that receives a super majority of voting weight $w(S_x(s,k),t,(l,\perp))$ from its voting set $S_x(s,k)$ for some round t initial vote $(l,\perp)$, there can not be a member $y \in S_v(r,k)$ that receives a super majority of voting weight from its voting set $S_y(s,k)$ for some round t vote $(\acute{l},\perp)$ with $l \neq \acute{l}$.*

Proof. The statement follows from the properties of a safe voting pattern. To see this in detail, first observe that if algorithm (7) is executed from $x \in S_v(r,k)$, the $\delta = 1$ branch is used to compute $x.vote(t)$, since $x.svp = v.svp \backslash \{r\}$ by proposition (5.14) and therefore $d_{x.svp}(s,t) = 1$. This however implies that the binary part of $x.vote(t)$ is undecided for all $x \in S_v(r,k)$.

We proof the statement by contradiction. Suppose that there is another member $y \in S_v(r,k)$ that receives a super majority of voting weight $w > w(S_y(s,k)) \ominus d_s$ for some vote $(\acute{l},\perp)$ with $l \neq \acute{l}$ from its voting set $S_y(s,k)$. Then x must receive more then $w(S_y(s,k)) \ominus 2 \cdot d_s$ voting weight for $(\acute{l},\perp)$, since $S_v(r,k)$ is a safe voting pattern, which implies $|w(S_y(s,k),t,(\acute{l},\perp)) \ominus w(S_x(s,k),t,(\acute{l},\perp))| \leq d_s$.

Moreover since the overall voting weight of $S_y(s,k)$ is strictly larger then $3 \cdot d_s$, x must receive more then d_s voting weight for $(\acute{l},\perp)$ and at the same time more then $w(S_x(s,k)) \ominus d_s$ voting weight for $(l,\perp)$, which is a contradiction.

To see that, let $S_x^{(l,\perp)}$ be the set of all members from $S_x(s,k)$ that vote for $(l,\perp)$ and $S_x^{(\acute{l},\perp)}$ the set of members from $S_x(s,k)$ that vote for $(\acute{l},\perp)$. Then

$$\begin{aligned} w(S_x^{(\acute{l},\perp)} \cap S_x^{(l,\perp)}) &= \\ w(S_x^{(\acute{l},\perp)}) \oplus w(S_x^{(l,\perp)}) \ominus w(S_x^{(\acute{l},\perp)} \cup S_x^{(l,\perp)}) &> \\ d_s \oplus w(S_x(s,k)) \ominus d_s \ominus w(S_x^{(\acute{l},\perp)} \cup S_x^{(l,\perp)}) &= \\ w(S_x(s,k)) \ominus w(S_x^{(\acute{l},\perp)} \cup S_x^{(l,\perp)}) &\geq 0 \end{aligned}$$

This means that there are members of $S_x(s,k)$ that have a vote both for $(l,\perp)$ and $(\acute{l},\perp)$ in round t, which is a contradiction, since each vertex has a single vote in any round only. □

Proposition 6.5 (Message presorting)**.** *Let v be a vertex with $|v.svp| \geq 3$ and $\{r > s > t\} \subset v.svp$ the three maximum elements from $v.svp$. Then there is a message l and each member of $S_v(r,k)$ has a round*

t vote either for $(l,\perp)$ or $(\oslash,\perp)$ and no other round t votes appear in $S_v(r,k)$.

Proof. Let $x \in S_v(r,k)$. Then algorithm (7) computes $\delta = 1$ if executed by x and x either received a super majority of initial voting weight for some round t vote $(l,\perp)$ from its voting set $S_x(s,k)$ or it does not. In the first case $x.vote(t) = (l,\perp)$ and in the second $x.vote(t) = (\oslash,\perp)$. Now suppose that x and y are both members of $S_v(r,k)$, that both received a super majority of voting weight for some round t vote $(l,\perp)$ and $(\acute{l},\perp)$, respectively. Then the previous proposition (6.4) implies $l = \acute{l}$. □

Proposition 6.6 (Graded Agreement). *Let v be a vertex with $|v.svp| \geq 4$, $\{r > s > t > u\} \subset v.svp$ the four maximum elements from $v.svp$ and let x and y be two members of $S_v(r,k)$. If x has round u vote $(l,1)$ or $(l,0)$ for some message $l \neq \oslash$ and y has round u vote $(\acute{l},1)$ or $(\acute{l},0)$ for some message $\acute{l} \neq \oslash$, then $l = \acute{l}$.*

Proof. This is our adaptation of the $g_i, g_j > 0 \Rightarrow v_i = v_j$ property in the definition of graded consensus and a consequence of the previous proposition.

To see that, first observe that if algorithm (7) is executed from $x \in S_v(r,k)$, the $\delta = 2$ branch is used to compute $x.vote(u)$, since $x.svp = v.svp\backslash\{r\}$ by proposition (5.14) and therefore $d_{x.svp}(s,u) = 2$.

Now, if x votes for $(l,0)$ or $(l,1)$ in round u with $l \neq \oslash$, it must have received more then d_s voting weight for $(l,\perp)$ from its voting set $S_x(s,k)$ and since $S_v(r,k)$ is a safe voting patter, we know $|w(S_x(s,k),(l,\perp)) - w(S_y(s,k),(l,\perp))| \leq d_s$ for all members $y \in S_v(r,k)$. Hence each member of $S_v(r,k)$ must have received at least some voting weight for $(l,\perp)$.

This implies that there must be a member of y's safe voting pattern $S_y(s,k)$ that has a round u vote $(l,\perp)$. But from the previous proposition (6.5) we know, that then no other member in y's safe voting pattern S_y can vote for some actual message $\acute{l} \neq l$. Therefore if y does not vote $(\oslash,\perp)$, it must have voted $(l,0)$ or $(l,1)$. □

Proposition 6.7 (Bounded grading). *Let v be a vertex with $|v.svp| \geq 4$, $\{r > s > t > u\} \subset v.svp$ the four maximum elements from $v.svp$ and let x be a members of $S_v(r,k)$ with $x.vote(u) = (l,0)$ for some message $l \neq \oslash$. Then there can not be a member $y \in S_v(r,k)$ with $y.vote(u) = (\oslash,1)$*

Proof. This is our adaptation of the $|g_i - g_j| \leq 1$ property in the definition of graded consensus. To start, observe that if algorithm (7) is executed from $x \in S_v(r,k)$, the $\delta = 2$ branch is used to compute $x.vote(u)$, since $x.svp = v.svp\backslash\{r\}$ by proposition (5.14) and therefore $d_{x.svp}(s,u) = 2$.

Now, for x to compute $x.vote(u) = (l,0)$, l must be an actual message, i.e $l \neq \oslash$ and x must have received a super majority of voting weight for $(l,\perp)$ from its voting set $S_x(s,k)$.

But then every other member $y \in S_v(r,k)$ must receive strictly more then d_s voting weight for $(l,\perp)$, since $S_v(r,k)$ is a safe voting pattern, which implies $w(S_y(s,k)) > 3 \cdot d_s$ as well as $|w(S_x(s,k),(l,\perp)) \ominus w(S_y(s,k),(l,\perp))| \leq d_s$. This however implies, that y's execution of (7) can not compute $y.vote(u) = (\oslash,1)$. □

Proposition 6.8 (Graded consistency). *Let v be a vertex with $|v.svp| \geq 4$, $\{r > s > t > u\} \subset v.svp$ the largest four elements and let there be a vertex $\acute{v} \in S_v(r,k)$, such the members $x \in S_{\acute{v}}(s,k)$ are in agreement on a round u vote $(l,\perp)$. Then each member $y \in S_v(r,k)$ computes its round u vote as $y.vote(u) = (l,0)$.*

Proof. For a first orientation, observe that the execution of algorithm (7) from $\acute{v} \in S_v(r,k)$, uses the $\delta = 2$ branch to compute $\acute{v}.vote(u)$, since $\acute{v}.svp = v.svp\backslash\{r\}$ by proposition (5.14) and therefore $d_{\acute{v}.svp}(s,u) = 2$. The same reasoning shows that the $\delta = 1$ branch is used to compute $x.vote(u)$ if algorithm (7) is executed from any $x \in S_{\acute{v}}(s,k)$.

Since the members of $S_{\acute{v}}(s,k)$ are in agreement on a round u vote $(l,\perp)$, by definition each member $x \in S_{\acute{v}}(s,k)$ receives a super majority of initial voting weight for $(l,\perp)$ from its voting set $S_x(t,k)$, which implies $x.vote = (l,\perp)$ for all $x \in S_v(s,k)$ by corollary (6.3).

Then $\acute{v}$ receives all voting weight $w(S_{\acute{v}}(s,k))$ for $(l,\perp)$ and no voting weight $w(S_{\acute{v}}(s,k),(\acute{l},\perp)) = 0$ for any other vote $(\acute{l},\perp)$ with $l \neq \acute{l}$. However since $S_v(r,k)$ is a safe voting pattern, each member

$y \in S_v(r,k)$ receives at most d_r voting weight for any $(\acute{l}, \perp)$ other then $(l, \perp)$ from its voting set $S_y(s,k)$, which implies, that the overall voting weight y receives for $(l, \perp)$ must be at least $w(S_y(s,k)) \ominus d_s$, since each member of $S_y(s,k)$ has exactly one round u vote. But then y's execution of (7) gives $\delta = 2$ and then $y.vote(u) = (l, 0)$. □

6.3.2 Binary byzantine agreement

As we have seen in the previous section, the $\delta \in \{0,1,2\}$ branches of algorithm (7) are an adaptation of Feldman & Micali's gradecast algorithm. In this section, we proof that the $\delta \geq 3$ branches simulate Micali's binary byzantine agreement protocol BBA^*, if we, for a moment, consider the binary part $(\cdot, b)$ of any vote (l, b) only. Our proofs are strongly influenced by Micali's original ideas as provided in [9].

Proposition 6.9 (Binary quorum intersection)**.** *Let v be a vertex with $|v.svp| \geq 5$, $\{r > s\} \subset v.svp$ the largest two elements of $v.svp$ and $x \in S_v(r,k)$ a member that receives a super majority of voting weight $w > w(S_x(s,k)) - d_s$ for some round u vote $(\cdot, 0)$ with $d_{x.svp}(s,u) \geq 4$ from its voting set $S_x(s,k)$. Then there is no member $y \in S_v(r,k)$, that receives a super majority of voting weight $w(S_y(s,k)) - d_s$ for a round u vote $(\cdot, 1)$ from its voting set $S_y(s,k)$ and vice versa.*

Proof. First observe that the execution of algorithm (7) from any $y \in S_x(s,k)$, uses a $\delta \geq 3$ branch to compute $y.vote(u)$, since $y.svp = x.svp \backslash \{s\}$ by proposition (5.14) and therefore $d_{y.svp}(t,u) \geq 3$ for $t = max(y.svp)$. Therefore the binary part of $y.vote(u)$ is decided for any $y \in S_x(s,k)$.

We proof the theorem by contradiction and assume that there is a member x of $S_v(r,k)$, that receives a super majority of voting weight $w > w(S_x(s,k)) \ominus d_s$ for a round u vote $(\cdot, 0)$ from its voting set $S_x(s,k)$ and another member $y \in S_v(r,k)$ that received a super majority of voting weight $w(S_y(s,k)) \ominus d_s$ for a vote $(\cdot, 1)$ in the same round through its voting set $S_y(s,k)$.

Since $S_v(r,k)$ is a safe voting pattern, both voting sets $S_x(s,k)$ and $S_y(s,k)$ have an overall weight strictly larger then $3 \cdot d_s$. Moreover $|w(S_x(s,k), (\cdot, 0)) \ominus w(S_y(s,k), (\cdot, 0))| < d_s$ implies that y must have received strictly more then $w(S_x(s,k)) \ominus 2 \cdot d_s$ voting weight for $(\cdot, 0)$.

Now, let S_y^0 and S_y^1 be the subsets of $S_y(s,k)$ through which y received votes for $(\cdot, 0)$ and $(\cdot, 1)$, respectively. Then $S_y^* := S_y^0 \cup S_y^1$ is again a subset of $S_y(s,k)$. If we use the identity $w(S) = w(S_1) \oplus w(S_2) \ominus w(S_1 \cap S_2)$ for the weights of a cover S_1 and S_2 of a set S we get

$$\begin{aligned} w(S_y^1 \cap S_y^0) = \\ w(S_y^1) \oplus w(S_y^0) \ominus w(S_y^*) > \\ w(S_y(s,k)) \ominus d_s \oplus w(S_x(s,k)) \ominus 2d_s \ominus w(S_y^*) = \\ \big(w(S_y(s,k)) \ominus w(S_y^*)\big) \oplus (w(S_x(s,k)) \ominus 3d_s) > 0 \end{aligned}$$

since S_y^* is a subset of $S_y(s,k)$ and the weight of $S_x(s,k)$ is strictly larger then $3d_s$. But no vertex can vote both for $(\cdot, 0)$ and $(\cdot, 1)$ in the same round. Hence we arrive at a contradiction. The proof for the vice versa case is exactly analog. □

Proposition 6.10. *Let v be a vertex with $|v.svp| \geq 5$, $\{r > s\} \subset v.svp$ the largest two elements from $v.svp$ and p_{coin} the probability that $b := lsb(H(\acute{v}_x.m))$ is the same for all $x \in S_v(r,k)$ and maximum weight vertex $\acute{v}_x \in S_x(s,k)$. If there is an element $u \in v.svp$ with $d_{v.svp}(s,u) \geq 3$ and $d_{v.svp}(s,u)$ mod $3 = 2$, then, with probability at least $p_{coin}/2$, all members $x \in S_v(r,k)$ will have the same vote $x.vote(u) = (\cdot, b)$ for a binary value $b \in \{0,1\}$.*

Proof. Since $x.svp = v.svp \backslash \{r\}$ for every $x \in S_v(r,k)$ by proposition (5.14), the requirement $d_{v.svp}(s,u)$ mod $3 = 2$ implies that the computation of x's round u vote in algorithm (7) executes the δ mod $3 = 2$ branch, e.g. the genuine-coin-flip stage. The quorum intersection theorem (6.9) then induces the following five exclusive cases:

1.) Every member $x \in S_v(r,k)$ receives a super majority of votes $(l, 0)$ for some message l and computes $x.vote(t) = (l, 0)$ in line (53). Hence every member votes $x.vote(\cdot, 0)$

2.) Every member $x \in S_v(r,k)$ receives a super majority of votes $(l, 1)$ for some message l and computes $x.vote(s) = (l, 1)$ in line (55). Hence every member votes $x.vote(\cdot, 1)$

3.) No member $x \in S_v(r,k)$ receives a super majority, neither for $(l,0)$ nor for $(\acute{l},1)$. Hence all members of $S_v(r,k)$ compute their vote as $(\cdot,b)$ in line (58) where b is the least significant bit $b := lsb(H(\acute{v}_x.m))$ of the vertex $\acute{v}_x \in S_x(s,k)$ that has the highest voting weight in $S_x(s,k)$. Then with probability p_{coin}, agreement will hold on $(\cdot,b)$, as the probability of b being the same for all $x \in S_v(r,k)$ is assumed to be p_{coin}.

4.) Some members $x \in S_v(r,k)$ receive a super majority of voting weight for some $(l,0)$ and some neither receive a super majority for $(\acute{l},0)$ nor for $(\tilde{l},1)$. Let $S_0 \subset S_v(r,k)$ be the set of members that receives a super majority of voting weight for $(l,0)$ and S_b the set of members that receives no super majority at all. Then all processes in S_0 execute line (53) and vote $(\cdot,0)$ and all processes in S_b execute line (58) and vote $(\cdot,b)$. Hence with probability at least $p_{coin}/2$, every member $x \in S_v(s,k)$ votes $x.vote(t) = (\cdot,0)$.

5.) Some members $x \in S_v(r,k)$ receive a super majority of voting weight for some $(l,1)$ and some neither receive a super majority for $(\acute{l},0)$ nor for $(\tilde{l},1)$. The argumentation is then analog to the previous situation.

From the quorum intersection theorem (6.9), we know that it is impossible for two members of the same safe voting pattern to receive super majorities both for $(\cdot,0)$ and $(\cdot,1)$. Hence the previous five case are exclusive and the proposition follows. □

Proposition 6.11 (Agreement stability)**.** *Let v be a vertex with $|v.svp| \geq 5$, $r = max(v.svp)$ the maximum element from $v.svp$ and suppose that there is an element $u \in v.svp$ with $d_{v.svp}(r,u) \geq 3$ such that the members $x \in S_v(r,k)$ are in agreement on some round u binary vote $(\cdot,b)$ with $b \in \{0,1\}$. If $\acute{v}$ is another vertex with $\acute{v}.svp \neq \emptyset$, $q = max(\acute{v}.svp)$ and $v \in S_{\acute{v}}(q,k)$, then every member $y \in S_{\acute{v}}$ has the same round u vote $y.vote(u) = (\cdot,b)$, too.*

Proof. We proof the proposition for $(\cdot,0)$. The situation for $(\cdot,1)$ is analog. Since the members of $S_v(r,k)$ are in agreement on a round u binary vote $(\cdot,0)$, corollary (6.3) implies, that every member $x \in S_v(r,k)$ computes $x.vote(u) = (\cdot,b)$. Hence all voting weight of the safe voting pattern votes for $(\cdot,0)$, e.g. $w(S_v(r,k),u,(\cdot,0)) = w(S_v(r,k))$ as well as $w(S_v(r,k),u,(\cdot,1)) = 0$ holds, as no vertex can have more then one vote in a round. But since $S_{\acute{v}}(q,k)$ is a safe voting pattern, we know $|w(S_v(r,k),u,(\cdot,1)) \ominus w(S_y(r,k),u,(\cdot,1))| < d_r$ for all $y \in S_{\acute{v}}$, which implies $w(S_y(r,k),u,(\cdot,1)) < d_r$. However each member of $S_y(r,k)$ either votes for $(\cdot,0)$ or $(\cdot,1)$, since ever member has exactly one vote in a round. This implies $w(S_y(r,k),u,(\cdot,0)) = w(S_y(r,k)) \ominus w(S_y(r,k),(\cdot,1)) < w(S_y(r,k)) \ominus d_r$. Hence y receives a super majority of voting weight for $(\cdot,0)$ and therefore votes $(\cdot,0)$. This is true for all $y \in S_{\acute{v}}(q,k)$. □

Proposition 6.12. *Let v be a vertex with $|v.svp| \geq 5$, $\{r > s\} \subset v.svp$ the highest two elements and let $u \in v.svp$ be a round number with $d_{v.svp}(r,u) > 3$, such that there is a member $x \in S_v(r,k)$ that receives a super majority of voting weight $w > w(S_x(s,k)) \ominus d_s$ from its voting set, for a round u vote $(\cdot,b)$ with $b \in \{0,1\}$ and $d_{v.svp}(s,u) \mod 3 = b$. Then all members $y \in S_v(r,k)$ vote $y.vote(u) = (\cdot,b)$.*

Proof. We proof the proposition for $(\cdot,0)$. The situation for $(\cdot,1)$ is analog.

In that case every member $x \in S_v(r,k)$ executes the $\delta \mod 3 = 0$, i.e. the coin-fixed-to-zero branch of algorithm (7). Since x received a super majority of voting weight $w > w(S_x(s,k)) \ominus d_s$ from its voting set $S_x(s,k)$ for a round u vote $(\cdot,0)$ and $S_v(r,k)$ is a safe voting pattern, no other member $y \in S_v(r,k)$ can receive a super majority of voting weight for a vote $(\cdot,1)$ due to the binary quorum interesection theorem (6.9). However this implies, that each member of $y \in S_v(r,k)$ computes its vote as $(\cdot,0)$ either according to line 31 or line 38. □

Proposition 6.13 (Eventual Agreement)**.** *Suppose that the probability for the appearance of new rounds and safe voting pattern is not zero. Let s be a round number, such that there is a safe voting pattern $S_{\acute{v}}(s,k)$ for some vertex $\acute{v}$ in round s. Then, with probability one, there will be a vertex v with $r = max(v.svp)$ and $s \in v.svp$, such that all members of $S_v(r,k)$ will be in agreement on the binary part of their round s votes, i.e. $x.vote(s) = (\cdot,b)$ holds for all $x \in S_v(r,k)$.*

Proof. As the probability of new rounds to appear is not zero and safe voting patterns will appear at least in some of these rounds, there will be vertices v, with $v.svp \neq \emptyset$, $s \in v.svp$ and $d_{v.svp}(max(v.svp), s)$ mod $3 = 2$. But then proposition (6.10) implies that the probability to reach agreement on some round s vote in $S_v(r,k)$ is not zero. Since there is an unbounded amount of those vertices, agreement holds eventually with probability one. □

6.3.3 Virtual leader agreement

Proposition 6.14 (Eventual Agreement)**.** *Suppose that the probability for the appearance of new rounds and safe voting pattern is not zero. Let s be a round number, such that there is a safe voting pattern $S_{\acute{v}}(s,k)$ for some vertex $\acute{v}$ in round s. Then, with probability one, there will be a vertex v with $r = max(v.svp)$ and $s \in v.svp$, such that all members of $S_v(r,k)$ will be in agreement on a round s vote, i.e. $x.vote(s) = (l,b)$ holds for all $x \in S_v(r,k)$ and message l.*

Proof. Due to proposition (6.13), we know that with probability one, there will be a vertex v with $r = max(v.svp)$ and $s \in v.svp$, such that all members of $S_v(r,k)$ will be in agreement on the binary part of a round s vote, i.e. $x.vote(s) = (\cdot, b)$ holds for all $x \in S_v(r,k)$ and some $b \in \{0,1\}$.

If binary agreement holds on $(\cdot,1)$, that is $x.vote(s) = (\cdot,1)$ for all $x \in S_v(r,k)$, then line (42) will be executed by every member of $S_v(r,k)$, hence each such member computes $x.vote(s) = (\oslash, b)$ and agreement holds on $\oslash$.

If binary agreement holds on $(\cdot,0)$, then $|v.svp| \geq 5$ and there is a $t \in v.svp$ such that $d_{v.svp}(t,s) = 3$. Then there is a vertex $\tilde{v}$ and a safe voting patter $S_{\tilde{v}}(t,k)$ in the past of v, such that at least one member must have received a super majority of voting weight for some vote $(l,\perp)$ with $l \neq \oslash$, because otherwise, all members of $S_{\tilde{v}}(t,k)$ would be in agreement on $(\cdot,1)$ and by proposition (6.11) stay in agreement on that vote, which contradicts our assumption, that agreement holds on $(\cdot,0)$.

Hence proposition (6.7) implies, that no member of $S_{\tilde{v}}(t,k)$ votes $(\oslash,1)$ and proposition (6.6) then implies that all members of of $S_{\tilde{v}}(t,k)$ either vote $(l,0)$ or $(l,1)$ for the same message l. In any case all members of that round are in agreement on the message l. Therefore l always receives the most voting weight in consecutive rounds (simply because there is no other choice) and hence agreement continous to hold on l, which implies that all members of $S_v(r,k)$ compute $x.vote(s) = (l,0)$ □

Proposition 6.15 (Agreement stability)**.** *Let v be a vertex with $v.svp \neq \emptyset$, $r = max(v.svp)$ the largest element from $v.svp$ and let there be an element $t \in v.svp$ with $d_{v.svp}(r,t) \geq 3$ and the members $x \in S_v(r,k)$ are in agreement on some message l, i.e $x.vote(t) = (l,b)$ with $b \in \{0,1\}$. If $\acute{v}$ is another vertex with $\acute{v}.svp \neq \emptyset$, $q = max(\acute{v}.svp)$ and $v \in S_{\acute{v}}(q,k)$, then every member $y \in S_{\acute{v}}$ has a vote $y.vote(t) = (l,b)$ in round t, too.*

Proof. This follows from the binary agreement stability (6.11) and proposition (6.14). □

6.4 Total Order

6.4.1 Leader stream convergence

Proposition 6.16. *Suppose that the probability for the appearance of new rounds and safe voting pattern is not zero, let $j \in \Pi$ be a honest process and r a round number. Then j will eventually have a Lamport graph G, such that the set* $\text{LEADER}_G(r)$ *is not empty.*

Proof. If there will never be a safe voting pattern in round r, algorithm (7) will eventually execute line (2) for some vertex and insert $(r,\oslash)$ into $\text{LEADER}_G(r)$. If on the other hand r has a safe voting patter, proposition (6.14) implies that with probability one there will eventually be a vertex that has a safe voting pattern, such that all members of that pattern are in agreement on a round r leader. In that case, execution of (7) will enter line (33) or line (44) and therefore elements are inserted into $\text{LEADER}_G(r)$. □

Proposition 6.17. *Let $j \in \Pi$ be a honest process that has a Lamport graph G, such that there are n different elements in the set* $\text{LEADER}_G(r)$*. Then there is a round s in G with an overall amount of voting weight w_s^G strictly larger then $3 \cdot n \cdot d_s$.*

Proof. First of all, function (8) ensures, that all elements in $\text{LEADER}_G(r)$ always have the same round number, because a new element (t, l) is inserted only, if there are no elements $(\acute{t}, \acute{l}) \in \text{LEADER}_G(r)$, that have higher deciding rounds $\acute{t} > t$. Moreover, once an element is inserted, all elements with lower deciding rounds are deleted. This implies that the massage part of different elements from $\text{LEADER}_G(r)$ must differ, but the round parts are always the same.

Thats being said, we proof the proposition in case there are two different elements in $\text{LEADER}_G(r)$ only. The general argumentation is analog. To see that, let (l, s) and $(\acute{l}, s)$ be different elements of $\text{LEADER}_G(r)$. Then we know that there must be two different vertices v and $\acute{v}$, that both have round s safe voting patterns $S_v(s, k)$ and $S_{\acute{v}}(s, k)$, such that v's execution of algorithm (7) inserted (l, s) and $\acute{v}$'s execution inserted $(\acute{l}, s)$ into $\text{LEADER}_G(r)$.

However due to the execution of line (32), or (43), v's safe voting pattern $S_v(s, k)$ is in agreement on (l, b) and $\acute{v}$'s safe voting pattern $S_{\acute{v}}(s, k)$ is in agreement on $(\acute{l}, b)$, e.g. all members $x \in S_v(s, k)$ vote $x.vote(r) = (l, b)$ and all members $\acute{x} \in S_{\acute{v}}(s, k)$ vote $\acute{v}.vote(r) = (\acute{v}, b)$ for some $b \in \{0, 1\}$. But since any vertex has a single vote in any round only, both voting sets must be disjoint. However $S_v(s, k)$ as well as $S_{\acute{v}}(s, k)$ are safe voting patterns and each has an overall amount of voting weight strictly larger then $3 \cdot d_s$. □

Theorem 6.18 (Leader convergence)**.** *Suppose that the probability for the appearance of new rounds and safe voting pattern is not zero and let $j \in \Pi$ be a honest process. Then j will have a series of Lamport graphs $G(t)$, such that the series of sets $\text{LEADER}_{G(t)}(r)$ converges to contain a single element only.*

Proof. Since new rounds appear, j will obtain a stream of messages, that extend the current Lamport graph. The time indexed Lamport graphs can therefore be seen as a sequence, such that each consecutive graph contains strictly more elements then the previous one. Despite the fact, that time is a continuous index. The theorem then follows from proposition (6.17), our assumption (8) on the boundary of the difficulty oracle and agreement stability.

To see this in detail, we proof the theorem by contradiction and assume that $\text{LEADER}_{G(t)}(r)$ does not converge to a single element for $t \to \infty$. Then proposition (6.16) implies that there is a parameter t_0, such that each set $\text{LEADER}_{G(t)}(r)$ contains at least two elements for all $t > t_0$.

Let t_1 be a time parameter, such that there are at least two elements (s, l) and $(s, \acute{l})$ in $\text{LEADER}_{G(t_1)}(r)$. Proposition (6.17) then implies that each element is decided by execution of algorithm (7) from a vertex v_1 and a vertex $\acute{v}_1$ both of which have disjoint safe voting pattern and the overall voting weight of round s is strictly larger in the Lamport graph $G(t_1)$.

Now since new rounds and safe voting patterns appear forever there must be a time t_2 and two vertices v_2 and $\acute{v}_2$ that have safe voting pattern $S_{v_2}(s_1, k)$ and $S_{\acute{v}_2}(\acute{s}_1, k)$, such that v_1 is in the safe voting pattern of v_2 and $\acute{v}_1$ is in the safe voting pattern of $\acute{v}_2$. By proposition (6.11) agreement then continuous to hold in these pattern, which implies that they are disjoint. Moreover v_1 can not be in the past of $\acute{v}_2$ and vice versa. Hence the entire history must be disjoint and therefore any round between $u_1 = min\{s_1, \acute{s}_1\}$ must have disjoint last vertices. This however implies that any Lamport graph $G(t)$ has voting weight $\sum_{j=s}^{u_1} w_j^{G(t)}/d_j > 6$ for all $t \geq t_2$.

However since $\text{LEADER}_{G(t)}(r)$ does not converge by assumption we can repeat the argument an unbounded amount of times, which implies $\sum_{j=s}^{u_i} w_j^{G(t_2)}/d_j > 6$ for arbitrary large round numbers u_i which violates our assumption (8) on the difficulty oracle bound. □

Corollary 6.19 (Leader stream convergence)**.** *Suppose that the probability for the appearance of new rounds and safe voting pattern is not zero and let $j, k \in \Pi$ be two honest process. Then their leader streams will converge.*

Proof. The previous theorem (6.19) implies that each leader set will converge to a single element, which implies that the leader stream of each honest process will converge and it remains to show that the elements in both leader streams are identical. This however follows from our message dissemination assumption (6.1), since Lamport graphs of honest processes eventually converge to contain the same elements. □

All previous proof are based on the assumption that the voting weight function is essentially unbounded and that the difficulty oracle can be estimated by assumption (8) only. However some implementations might be much simpler, in that they don't have unbounded weight function or a strict upper bound on the voting can be computed. As the following corollary shows, this leads to much faster and cleaner convergence of the global leader stream.

Corollary 6.20 (Bounded voting weight leader stream)**.** *Let d be a difficulty oracle, G a Lamport graph and* $\text{LEADER}_G(\cdot)$ *the leader stream of G, such that the overall amount of voting weight in any round r is always in between $3d_r < w \leq 6d_r$. Then* $\text{LEADER}_G(r)$ *will never contain more then one element.*

Proof. This follows directly from proposition (6.17) as more then one element would imply that there is a round with overall voting weight strictly larger then $6d_r$. □

6.4.2 Total order convergence

We proof that Moser & Melliar-Smith's properties of a byzantine resistant total order algorithm as defined in (6) are satisfied, provided our set of assumptions (6.1) holds.

Proposition 6.21 (Partial Correctness)**.** *The asymptotically convergent total orders determined by any two non byzantine processes are consistent; i.e., if any non byzantine process has a Lamport graph that determines $v.total\_position = i$, then no honest process has a Lamport graph that determines $\acute{v}.total\_position = i$, where $\acute{v} \not\equiv v$.*

Proof. The theorem follows, since the leader streams of two honest processes eventually converge with probability one, by corollary (6.19) and the order is derived deterministically from the past of any element in the leader stream only. However that is equivalent among all Lamport graphs, by the equivalence of the past theorem (3.7). □

Proposition 6.22 (Consistency)**.** *The total order determined by any non byzantine process is consistent with the partial causality order; i.e. $\acute{v} \leq v$ implies $\acute{v}.total\_position \leq v.total\_position$.*

Proof. Let v and $\acute{v}$ be two vertices in a Lamport graph with $\acute{v} \leq v$, such that the total order of both vertices is not $\perp$. Let v be in the order cone of some momentrary round leader v_l, e.g. $v \in Ord(v_l)$. Then either $\acute{v} \in Ord(v_l)$ or not. In the former case both $v.total\_order$ and $\acute{v}.total\_order$ are computed by a topological sorting algorithm hence $\acute{v} \leq v$ implies $\acute{v}.total\_order \leq v.total\_order$ by the very properties of topologic sorting. In the latter case $\acute{v} \leq v$ implies $\acute{v} \in G_{v_l}$, but since $\acute{v} \notin Ord(v_l)$, we know $\acute{v}.total\_order < w.total\_order$ for all $w \in Ord(v_l)$, since the total order of all elements from $Ord(v_l)$ starts with a value greater then all previous totally ordered elements. □

Proposition 6.23 (Probabilistic Termination I)**.** *The probability that a honest process j computes $v.total\_position = i$ for some Lamport graph G, position i and vertex v increases asymptotically to unity as the number of steps taken by j tends to infinity.*

Proof. By theorem (6.19) every round r will converge to a single round leader l_r and at least some of these rounds will converge to a leader $\neq \oslash$ due to our initial vote assumption (6.1). This implies that the order loop (9) will execute some topological sorting, like (10), that assigns a total order position to all elements in the past of a leader vertex v with $v.m = l_r$ in any Lamport graph. As the leader converges, so does the order in its past and as this goes on forever there will eventually be a round u and a leader l_u, such that $|G_v| > i$ for v with $v.m = l_u$ and any $i \in \mathbb{N}$. Hence Lamport graph G will have a vertex with $v.total\_order = i$ and this vertex converges to a fixed value. □

Proposition 6.24 (Probabilistic Termination II)**.** *For each message m broadcast by a non byzantine process j, the probability that a non byzantine process k places some vertex v with $v.m = m$ in the total order, increases asymptotically to unity as the number of steps taken by k tends to infinity.*

Proof. Let m be a message and v a vertex with $v.m = m$ and round number r. Since new rounds appear forever, there will eventually be a round r and round r leader vertices v_l in the future of v, (i.e $v \in G_{v_l}$) that converge to a single element. As the past of these leaders gets ordered by algorithm (10) and this order converges, the order of v converges too. $\square$

7 Conclusion & Outlook

We have developed a family of total order algorithms that may survive in unstructured Peer-2-Peers networks and in the presence of momentary large amount of partitioning and faulty behavior. The system chooses different strategies with respect to Brewers CAP-theorem. If no forking occurs, it choose a strict availability & consistency strategy which allows for short finality. However if partition occurs, incarnated in the form of more then one safe voting pattern in a round, availability remains but consistency becomes probabilistically convergent only.

In contrast to most other approaches, our system is able to incorporate a Proof-Of-Work based voting strategy, which circumvent serious Proof-Of-Stake problems, like bootstrapping and runaways. Proof-Of-Work is difficult to use in byzantine agreement, because such a weight function is usually unbounded.

In any case, future research has to be made in finding optimal system parameter, like the difficulty oracle, incentivation & punishment, weight systems and the quorum selector. Of course its possible to just be creative and make these function up, but the author believes that a systematical search to find optima is much more reasonable.

However, if you would like to support the continuous production of content like this, please donate via one of the channels mentioned on the title page, or contact the author for additional solutions.

References

[1] Alchieri, E. A.; Bessani, A.N.; Fraga, J.S.; Greve, F.: *(2008) Byzantine Consensus with Unknown Participants. In Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS '08). Springer-Verlag, Berlin, Heidelberg, 22-40.*

[2] Baird, L.: *(2016) The Swirlds hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance, Swrirlds tech report, SWIRLDS-TR-2016-01.*

[3] Beck A.: *(2002) Hashcash - A Denial of Service Counter-Measure*

[4] Chen, J.; Micali, S.: *(2016). Algorand*

[5] Chevalier, P.; Kaminski, B.; Hutchison, F.; Ma, Q.; Sharma, S.: *(2018) Protocol for Asynchronous, Reliable, Secure and Efficient Consensus (PARSEC).*

[6] Danezis, G.; Hrycyszyn, D.: *(2018) Blockmania: from Block DAGs to Consensus.*

[7] Feldman P., Micali S.: *(1997) An Optimal Probabilistic Algorithm for Synchronous Byzantine Agreement. (Preliminary version in STOC 88.) SIAM J. on Computing.*

[8] Lamport, L.: *(1978) Time, clocks, and the ordering of events in a distributed system, Commun. Assoc. Comput. Mach. 21, 558-565.*

[9] Micali, S.: *(2018). Byzantine Agreement , Made Trivial*

[10] Louise E. Moser and P. M. Melliar-Smith: *(1999) Byzantine-Resistant Total Ordering Algorithms. Inf. Comput.. 150. 75-111.*

[11] Nakamoto, S.: *(2009) Bitcoin: A Peer-to-Peer Electronic Cash System.*

[12] Spivak, D. I.: *(2014) Category Theory for the Sciences. MIT Press.*