Synchronous Consensus During Incomplete Synchrony

Ivan Klianev
Ivan.Klianev@gmail.com
Transactum Pty Ltd
Sydney, Australia

ABSTRACT
We present an algorithm for synchronous deterministic Byzantine consensus, tolerant to links failures and links asynchrony. It cares for a class of networks with specific needs, where both safety and liveness are essential, and timely irrevocable consensus has priority over highest throughput. The algorithm operates with redundant delivery of messages via indirect paths of up to 3 hops, aims all correct processes to obtain a coherent view of the system state within a bounded time, and establishes consensus with no need of leader. Consensus involves exchange of $2 \times n^3$ asymmetrically authenticated messages and tolerates up to $< n/2$ faulty processes.

We show that in a consensus system with known members: 1) The existing concepts for delivery over a fraction of links and gossip-based reliable multicast can be extended to also circumvent asynchrony and thereby convert the reliable delivery into a reliable bounded delivery. 2) A system of synchronous processes with bounded delivery does not need a leader – all correct processes from connected majority derive and propose the same consensus value from atomically consistent individual views on system’s state. 3) The required for bounded delivery asymmetric authentication of messages is sufficient for safety of the consensus algorithm. Key finding: the impossibility of safety and liveness of consensus in partial synchrony [26] is not valid in the entire space between synchrony and asynchrony. A system of synchronized processes, which communicate with asymmetrically authenticated messages over a medium susceptible to asynchrony and faults, can operate with: 1) defined tolerance to number of asynchronous links and/or faulty links per number of stop-failed and/or Byzantine processes; 2) leaderless algorithm with bounded termination; and 3) conceptually ensured simultaneous safety and bounded liveness.

CCS CONCEPTS
• Computing methodologies → Distributed computing methodologies; • Computer systems organization → Dependable and fault-tolerant systems and networks.

KEYWORDS
synchronous consensus, partial synchrony, synchronous consensus in partial synchrony, consensus over communication medium susceptible to faults and asynchrony

1 INTRODUCTION
1.1 Theoretical Background
Deterministic synchronous consensus has conceptual significance for the purposes of atomicity of committed distributed database transactions [44] [25]. It is impossible with a large fraction of all communication links being faulty [27]. With all links being correct and process faults up to a fraction of all processes, it was solved with an algorithm where the decision making takes as input a vector, having an element for each process, loaded with values received from all processes [47] [37]. Under the same limit of tolerated faulty links, it was also solved with a small fraction of all one-way links being faulty [16]. Aspects of synchronous consensus under simultaneous process and link failures were studied and solved in various contexts and scenarios [43] [55] [50] [52] [49] [51] [8]. Yet none of these works considers possible communication asynchrony.

Consensus cannot be solved with a single unannounced failure of a process under full asynchrony of processes, communication, and message ordering [23] and also under full asynchrony of either processes or communication [17]. Yet the needs of practice compel more attention to communication that is neither fully asynchronous nor fully synchronous. Its name partial synchrony [21] reflects the reality where intervals of full synchrony with unknown duration are followed by intervals of infringed full synchrony. First dealing with it was under the model of eventual synchrony [21], meaning eventual in time synchrony. This model stoically accepts the inevitability of intervals with asynchrony and allows termination only during large enough intervals of full synchrony. [20] [4]. Handling asynchrony in a proactive manner makes more sense. One, and so far the most popular, way of doing that is to give up the reliance on ability of correct processes to obtain coherent view of the system state for computation of consensus value and replace it with reliance on a leader to propose the value [13]. This approach is a tradeoff where vulnerability to stop-fail or Byzantine faults [37] of the leader replaces the vulnerability to communication asynchrony. A Byzantine-faulty leader can delay consensus without revealing itself. Hence, with this way of proactive handling of asynchrony, as well as with the passive waiting for its disappearance, consensus is time vulnerable. These outcomes led to conjecture of impossibility to ensure simultaneous safety and liveness of consensus systems under partial synchrony [26]. Leaderless consensus under partial synchrony is considered theoretically possible [10]. Yet at present all deterministic algorithms that solve Byzantine consensus over communication medium susceptible to asynchrony are leader-based [1] [12] [6] [45] [46] [11] [56], hence time-vulnerable.

Definition 1 (Leaderless Consensus Algorithm): Every process, from a quorum of correct processes with atomically consistent [31] views of system’s state, proposes the same value for consensus. Consensus is key for ensured atomic operations across database management systems [28]. These systems implement and use local transactions for consistency of operations and integrity of local data. Distributed transactions across these systems aim the same outcome and also by using local transactions, yet executed under external coordination. It involves use of a protocol from a family of distributed commit protocols [27] [40] [39], known for susceptibility to blocking [7] [57] and/or possibility to leave a database system participating in a distributed transaction in inconsistent state.
Definition 2 (Decentralized Database System): Data replicated on multiple database management system peers in different trust domains and modified with decisions made by quorum of peers.

Decentralized database systems solve half of the problem solved with distributed transactions – the stretch over distance – with global replication of locally executed transactions. These solutions ensure atomic (linearizable) [31] consistency across local systems’ data with state machine replication [35] [54] [38]. Users can safely transact across continents, but only within a decentralized database silos. Solving the second half requires non-blocking transactions across the boundaries of decentralized database systems.

For the duration of a distributed transaction, all participants performing a role in it (the transacting parties and the coordinator) operate as a single system. Its partitioning, or spoiled liveness of a participant, normally spoils its liveness and may compromise its safety, regardless of non-compromised safety of all participants, and the transaction cannot fulfill its purpose to ensure consistent outcome. Hence, distributed transaction system’s prerequisites for safety are: 1) non-partitioning; and 2) liveness and safety of all participants. Further on in the paper we elaborate on and present a way to tame the root cause for partitioning and spoiled liveness.

1.2 Problem Formulation

Root cause for the present inability of deterministic Byzantine consensus systems to operate with ensured safety and bounded liveness under partial synchrony is the perceived impossibility for bounded delivery during incomplete communication synchrony.

1.3 Motivation

Possibility for deterministic Byzantine consensus systems with ensured safety and bounded liveness in partial synchrony would make feasible distributed transactions between decentralized database systems to operate with consistency resilient to:
- Partitioning of the distributed transaction system; or
- Fraudulently ‘lost’ liveness by one participant.

1.4 Solution Brief

A digitally signed [48] message proves its authenticity to anyone with access to signing party’s public key. So, it might be relayed multiple times before reaching its final destination, in effect as if received directly from the sender. The key idea is to circumvent faulty and/or asynchronous links with epidemic dissemination [9] of messages and attain performance of a system with fault-free and asynchrony-free links. The goal is all correct processes to obtain a coherent view of the system state within a bounded time and establish consensus with no need of leader. From engineering perspective the solution is a trade-off: giving up the max throughput and efficient scalability in exchange for ensured bounded liveness.

1.5 Similar Works

This solution exhibits a major distinction from all previous works. It circumvents asynchronous links with bounded delivery over the links that operate synchronously. No known concept for delivery over a fraction of links considers links asynchrony or tolerates a non-negligible amount of it. Similarly, no known gossip-based concept or algorithm is concerned with bounded delivery.

Synchronous Consensus Over a Fraction of Links

Siu et al [55] presents a concept for fault-tolerant virtual channels in a synchronous deterministic Byzantine consensus system, where processes and links can fail independently and simultaneously. It ensures reliable delivery for the purposes of synchronous consensus over a fraction of links via circumvention of faulty link. This concept does not consider asynchrony of links.

Aguilera et al [2] [3] demonstrates a case of solved synchronous deterministic Byzantine consensus with a small fraction of all links operating synchronously. It is rather an exception from the normal distribution of asynchrony – one correct process having all inbound and outbound links synchronous. The number of permutations with such outcome shows this case’s negligibly small probability.

Consensus via Gossip-based Reliable Multicast

In a distributed system where every process knows every other, the algorithm of Birman et al [9] uses gossips, a.k.a. epidemic dissemination of messages, to deliver to all recipients, even if the sender fails before sending to all. The algorithm tolerates process crash failures and a bounded number of soft failures of links, such as buffer overflow. The algorithm does not consider Byzantine failures, asynchrony of links, or bounded delivery.

1.6 Contribution

This paper contributes on the subject matter of consensus in partial synchrony:
1. The algorithm for synchronous deterministic consensus with highest known tolerance to communication asynchrony;
2. The first leaderless consensus algorithm over a communication medium susceptible to asynchrony and faults;
3. Proof of possibility for deterministic Byzantine consensus systems to operate with ensured safety and bounded liveness.

1.7 Content

The rest of the paper is structured as follows: Section 2 contains thorough formal presentation of the concept. Section 3 presents the system model of distributed computing. Section 4 presents the algorithm. Section 5 proves the claims of this paper. Section 6 summarizes the results. Section 7 concludes the paper. Appendix shows assessment of tolerance with simulation and regression analysis.

2 THE CONCEPT

A digitally signed [48] message might be relayed multiple times before reaching its final destination, in effect as if received directly from the sender. Thus, in spite of faults and/or asynchrony of some links, a message might reach all recipients via indirect channels – virtual aggregations of multi-hop indirect delivery paths.
2.1 Direct Channels

Distributed consensus system $S$ of $N$ processes $(N \geq 3)$ has a full set $C$ of $(N-1)$ direct channels $c_{pq}$ $(1 \leq p \leq N, 1 \leq q \leq N, p \neq q).$ Channel $c_{pq}$ represents a one-way link for delivery from process $p$ to process $q.$ Channel value 1 denotes correct and synchronous link. Value 0 denotes faulty or asynchronous. Matrix $C$ represents the full set $C$ and also $N$ values representing $c_{pq}$ $(1 \leq p \leq N, 1 \leq q \leq N, p \equiv q),$ i.e. a link of every process to itself, always with value 1:

$$C = \begin{pmatrix}
 c_{11} & c_{12} & \ldots & c_{1N} \\
 c_{21} & c_{22} & \ldots & c_{2N} \\
 \vdots & \vdots & \ddots & \vdots \\
 c_{N1} & c_{N2} & \ldots & c_{NN}
\end{pmatrix}$$

(1)

2.2 Two-hop Indirect Delivery Paths

System $S$ has also a full set $C'$ of two-hop indirect delivery paths comprising $N \times (N - 1) \times (N - 2)$ individual paths, as every direct path has $(N-2)$ two-hop alternatives. A message from process $p$ to process $q$ can be delivered with two hops, the first from process $p$ to process $r$ and the second from process $r$ to process $q.$ Indirect delivery over two links with bounded delivery is longer but still bounded. A two-hop indirect delivery path is denoted in two ways:

- $c_{pqr}'(1 \leq p \leq N, 1 \leq q \leq N, 1 \leq r \leq N, p \neq q, r \neq p, r \neq q),$ indicating delivery from process $p$ to process $q$ via process $r$; or

- $c_{pqr}''$ indicating delivery over any of the two-hop indirect delivery paths from process $p$ to process $q,$ represented by a vector of $(N-2)$ elements, each representing a vector of 2 elements: $c_{pr}$ and $c_{qr}.$

Value of $c_{pqr}$ is 1 if both $c_{pr}$ and $c_{qr}$ have value 1. Value of $c_{pqr}'$ is 1 if among the existing $(N-2)$ indirect delivery paths there exists a path where both $c_{pr}$ and $c_{qr}$ have value 1.

Formal Matrix $C'$ represents a full set of two-hop indirect delivery paths:

$$C' = \begin{pmatrix}
 c'_{11} & c'_{12} & \ldots & c'_{1N} \\
 c'_{21} & c'_{22} & \ldots & c'_{2N} \\
 \vdots & \vdots & \ddots & \vdots \\
 c'_{N1} & c'_{N2} & \ldots & c'_{NN}
\end{pmatrix}$$

(2)

where value of $c'_{pq}(p = q)$ is always 0 as it has no meaning and where value of $c'_{pq}(p \neq q)$ is defined by vector $c_{pq}:'$

$$c'_{pq} = \{ k_{1}' \quad k_{2}' \quad \ldots \quad k_{N-2}' \}$$

(2.1)

comprising all two-hop paths from $p$ to $q,$ and computed as:

$$c'_{pq} = k_{1}' \lor k_{2}' \lor \cdots \lor k_{N-2}'$$

(2.2)

where $k_{i}' = c'_{pqr},$ and value of $c'_{pqr}$ is defined by vector $C'_{pqr}:

$$c'_{pqr} = \{ k_{1} \quad k_{2} \}$$

(2.3)

specifying the links participating in the path, and computed as:

$$c'_{pqr} = k_{1} \land k_{2}$$

(2.4)

2.3 Three-hop Indirect Delivery Paths

System $S$ has as well a full set $C''$ of three-hop indirect delivery paths comprising $(N-1) \times (N-2) \times (N-3)$ paths as every two-hop indirect path has $(N-3)$ three-hop alternatives. A message from process $p$ to process $q$ can be delivered with three hops: the first from $p$ to process $r,$ the second from $r$ to process $s,$ and the third from process $s$ to $q.$ A three-hop indirect delivery path is denoted also in two different ways:

- $c_{pqr}'''(1 \leq p \leq N, 1 \leq q \leq N, 1 \leq r \leq N, 1 \leq s \leq N, p \neq q, r \neq p, r \neq s, s \neq p, s \neq q, s \neq r),$ indicating delivery from $p$ to $q$ via processes $r$ and $s$; or

- $c_{pqr}'''$ indicating any of the three-hop indirect delivery paths from $p$ to $q,$ represented by a matrix of $(N-2) \times (N-2)$ elements, each representing a vector of 3 elements: $c_{pr}, c_{rs},$ and $c_{sq}.$

Value of $c_{pqr}'''$ is 1 if each of its constituent elements $c_{pr}, c_{rs},$ and $c_{sq}$ has value 1. Value of $c_{pqr}'''''$ is 1 if among the existing $(N-2) \times (N-2)$ indirect delivery paths there exists a path where each of its constituent elements $c_{pr}, c_{rs},$ and $c_{sq}$ has value 1.

Formal Matrix $C''$ represents full set of three-hop indirect delivery paths:

$$C'' = \begin{pmatrix}
 c''_{11} & c''_{12} & \ldots & c''_{1N} \\
 c''_{21} & c''_{22} & \ldots & c''_{2N} \\
 \vdots & \vdots & \ddots & \vdots \\
 c''_{N1} & c''_{N2} & \ldots & c''_{NN}
\end{pmatrix}$$

(3)

where the value of $c''_{pq}(p = q)$ is always 0 as it has no meaning and the value of $c''_{pq}(p \neq q)$ is defined by matrix $C_{pq}'''$, where $M = N - 2$:

$$C_{pq}''' = \begin{pmatrix}
 k_{1}' & k_{1}'' & \ldots & k_{1}''' \\
 k_{2}' & k_{2}'' & \ldots & k_{2}''' \\
 \vdots & \vdots & \ddots & \vdots \\
 k_{M1}' & k_{M2}'' & \ldots & k_{MM}'''
\end{pmatrix}$$

(3.1)

comprising all $M \times M$ three-hop paths from $p$ to $q,$ and computed as:

$$c_{pq}''' = k_{1}' \lor k_{1}'' \lor \cdots \lor k_{1}'''$$

(3.2)

$$k_{2}' \lor k_{2}'' \lor \cdots \lor k_{2}'''$$

$$\vdots$$

$$k_{M1}' \lor k_{M2}'' \lor \cdots \lor k_{MM}'''$$

(3.3)

where value of $k_{ij}'' = c_{pqr}'''(r = s)$ is 0 as it has no meaning and where value of $k_{ij}''' = c_{pqr}'''(s = r)$ is defined by vector $C_{pqr}'''$:

$$C_{pqr}''' = \{ k_{1} \quad k_{2} \quad k_{3} \}$$

(3.4)

For instance, value of $c_{1234}''' = c_{13} \land c_{34} \land c_{42}.$

2.4 Subsets of Channels

At any point of time, only a fraction of links in $S$ may be correct and operate synchronously. Yet, $S$ may have $D$ delivery subsets $C_{D}(1 \leq i \leq D),$ where the channels of every subset $C_{D,i}$ can deliver all messages, directly or indirectly, within the synchrony bound. Given that all $N$ processes in $S$ are correct, a subset of $C$ (1) is a delivery one if it can deliver a message from a process $p$ to any other process $q$ over $c_{pq}$ $(1 \leq p \leq N, 1 \leq q \leq N, p \neq q)$ (1) or over $c'_{pq}$ (2) or over $c''_{pq}$ (3) of the subset. A non-delivery subset is one where no $c_{pq}$ and no $c'_{pq}$ and no $c''_{pq}$ can deliver the message.
Existence of at least one delivery subset $Cd_i$ is the necessary and sufficient condition for delivery of all messages. Hence, the delivery of all messages can be prevented only by a number of faulty direct channels that renders all subsets non-delivery.

2.5 Effect of Faulty Processes

In a consensus system where any third party can verify authenticity of messages, a Byzantine-faulty process may deceive about its value, but may not tamper any message from other process and send it as authentic [47]. Its options are limited to failure to send a protocol message to some or all processes, or to deliberately delay its sending [19]. Hence in a system that relies on indirect delivery, a Byzantine-faulty process can do no more harm than a stop-failed process and consensus safety requires $N = 2F + 1$ processes, where $F$ is the number of tolerated stop-failed and/or Byzantine-faulty processes. The indirect delivery depends on strict execution of an algorithm. A process can spoil an indirect path that relies on its participation. Thus, a faulty process may convert some of the delivery to non-delivery subsets. Consequently, system $S$ may have $A$ subsets of channels $Ca_i (1 \leq i \leq A)$ called agreement subsets, where the channels in every subset $Ca_i$ deliver all messages necessary for reaching an agreement, directly or indirectly. Agreement subset of $C(1)$ is a subset that can synchronously deliver a message from any process $p$ of a group of at least $F + 1$ correct processes to any other process $q$ of the same group over $c_{pq} (1 \leq p \leq N, 1 \leq q \leq N, p \neq q)$ (1) or over $c_{pq}^\prime (2)$ or over $c_{pq}^\prime\prime (3)$ of the subset. A non-agreement subset is one where neither $c_{pq}, c_{pq}^\prime$, or $c_{pq}^\prime\prime$ of the subset can deliver the message synchronously.

2.6 Virtual Channels for Indirect Delivery

A set of Virtual indirect channels is a construction of links and processes interacting according to a deterministic protocol. Purpose of an indirect channel is to circumvent a faulty or asynchronous link via indirect synchronous delivery. A set of 2-hop indirect channels operates with one process that helps with two hops. The helper multicasts a message received from sender to $(N - 2)$ recipients, thus enacting $(N - 2)$ channels; A set of 3-hop indirect channels operates with $1 + (N - 2)$ helper processes. The first helper enacts $(N - 2)$ 2-hop indirect channels. The group of $(N - 2)$ second helpers enacts the third hop where each of the second helpers multicasts the received message to $(N - 3)$ recipients, on its first receiving.

3 SYSTEM MODEL

Critical for the concept is to minimize the chance that all inbound links to a process can together fail or become asynchronous. Failure of a link is typically due to network card’s overflown receive buffer. Avoiding it requires hardware with capacity for overlapped receiving of messages with max size from all peers. So, an avalanche of faulty links triggered by a single link fault can be ruled out.

Asynchrony of links is beyond Internet users’ ability to tame it. Multiple inbound links to a process inevitably share a router that may become overloaded. Fortunately, TCP congestion avoidance [33] algorithms use random early detection [32] [24] [34] to halve the TCP window of individual flows one-by-one when needed, thereby ruling out simultaneous switch to unbounded delivery.

3.1 Model Assumptions

Computation:
- Deterministic processes. Processes are implemented as computer code with state-defined deterministic execution on pre-selected CPU cores to run process threads.
- Deterministic protocol. Processes communicate with exchange of messages, which follows a deterministic protocol.

Communication and messages:
- Synchronous with limited exceptions. Messages are normally delivered within a known bounded amount of time.
- Signed. Messages are signed with third party verifiable digital signatures, i.e. able to prove its authenticity to every process.

Processors and processes:
- Synchronous. Processes execute on CPUs with a fixed upper bound on how faster one complete a processing cycle than another and on the difference in cache size and cache latencies.
- Synchronized. Processes start a consensus round with a max lag bound to less than RTTB/2 (round trip transmission bound), which is MDTB (message delivery time bound).
- No clocks drift. Synchronized processes stay synchronized via periodic re-synchronization of clocks.
- Known public keys. Every process knows the public key of every other process’ private key for signature.

Communication failures and asynchrony:
- Uncorrelated failures. A fault in one link is independent from a fault in another link, i.e. a failure in an inbound link cannot trigger an avalanche of inbound link failures.
- Uncorrelated asynchrony. Asynchronous operation of one link is independent from asynchronous operation of another link, i.e. an asynchronous inbound link cannot trigger an avalanche of asynchronous inbound links.

3.2 The System

The model considers a consensus system S comprising $N$ deterministic processes $(N \geq 3, N \geq 2F + 1)$, where $F$ is the tolerated number of faulty processes. Each process $P$ is uniquely identified within S with a system index $I (1 \leq I \leq N)$. The processes are connected by point-to-point one-way links. Every pair of processes is connected with a pair of links $c_{IsIr}$, where $Is$ identifies the sender process and $Ir$ identifies the receiver process. Existence of links where $Is = Ir$, i.e. from every process to itself, is assumed for simpler presentation. Every process $P$ has special memory locations: input register $R^In$ and output register $R^{Out}$. Register $R^In$ can contain up to $x$ bytes in size. Register $R^{Out}$ is a vector of $N$ elements $R^{Out}$:

$$R^{Out} = \{R^I_1, R^I_2, \ldots, R^I_N\} \quad (4)$$

Every process $P$ also has an unbound amount of internal storage and an allocated in its vector $E$ of $N$ extended digital signatures:

$$E = \{e_1, e_2, \ldots, e_N\} \quad (4.1)$$
where $e_i$ is extended digital signature over $R_i^{in}$ in form of a tuple of 2 elements: 1) $h_i$ is output of execution of the Hash function used in $S$ over $R_i^{in}$; and 2) $s_i$ is the signature of process $P_i$ over $h_i$.

The values in $R^{in}, R^{Out}$, and the internal storage of the processes of $S$ define the internal state of $S$. Initial states prescribe fixed starting value for all, in particular $R^{in}$ and $R^{Out}$ start with value 0 of every byte. States in which $R^{Out}$ contains $\geq (F + 1)$ elements with values $\neq 0$ are distinguished as decision states.

Each process $P$ acts deterministically according to a transition function $\Phi$, which cannot change the value of $R^{Out}$ once the process has reached a decision state; i.e., $R^{Out}$ is 'write-once'. System $S$ is specified by the function $\Phi$ and the values in $R^{in}$ of every process.

Processes communicate with signed messages. A message is tuple $(T, Is, Id, Io, m, Do)$, where $T$ is a constant specifying message type, $Is$ is index of the source process, $Id$ is index of the destination process, $Io$ is index of the originator process, $m$ is 'message value' from a fixed universe $M$, and $Do$ is digital signature of the originator process over $m$. Where $Io \neq 1$, this indicates that the message $m$ sent by process $P_i$ contains the message and the digital signature that $P_i$ has received from $P_{Io}$.

Communication within a consensus round happens in two phases. During Phase One each process $P_i$ transmits to every process $P_{Id}(1 \leq Id \leq N, Id \neq Is)$, the value in its input register $R_{Id}^{in}$ and its signature over the value in $R_{Id}^{in}$. During Phase Two each process $P_i$ transmits the content of its $E_{Is}$ (4.1) and its signature over $E_{Is}$ to every process $P_{Id}(1 \leq Id \leq N, Id \neq Is)$.

At the Phase One beginning, state of $S$ is defined by vector $V_1$:

$$V_1 = \{o_1, o_2, \ldots, o_{1N}\} \quad (5)$$

An element $o_i(1 \leq i \leq N)$ contains the value $R_i^{in}$ of process $P_i$ and its signature $s_i$ over $R_i^{in}$. During Phase One process $P_i$ sends $o_i$ to every process $P_{Id}(1 \leq j \leq N, j \neq i)$ over $c_{ij}$ (1). Matrix $D1$ presents what was delivered by $S$ during Phase One. An element $d_{ij}^{1}$ is the value of $o_1$, as delivered over $c_{ij}$, i.e. where value of $c_{ij}$ is 0, indicating a faulty direct channel, the delivered value $d_{ij}^{1}$ is 0; otherwise $d_{ij}^{1} = o_i$:

$$D1 = \begin{bmatrix}
  d_{111} & d_{112} & \ldots & d_{11N} \\
  d_{121} & d_{122} & \ldots & d_{12N} \\
  \vdots & \vdots & \ddots & \vdots \\
  d_{1N1} & d_{1N2} & \ldots & d_{1NN}
\end{bmatrix} \quad (5.1)$$

On receiving $d_{ij}^{1} \neq 0$ from process $P_i$, process $P_j$ executes the system’s Hash function over $R_i^{in}$ component of $d_{ij}^{1}$, writes the outcome as $h_i$ component of element $e_i$ of its vector $E_i$, uses the signature $s_i$ component of $d_{ij}^{1}$ to verify the signature, writes $s_i$ as the $s_i$ component of element $e_i$ of its vector $E_i$, and in case of correctly verified signature writes $R_i^{in}$ in its vector $R^{Out}$ (4).

The values in $R^{in}$, $R^{Out}$, and the internal storage of the processes of $S$ define the internal state of $S$. Initial states prescribe fixed starting value for all, in particular $R^{in}$ and $R^{Out}$ start with value 0 of every byte. States in which $R^{Out}$ contains $\geq (F + 1)$ elements with values $\neq 0$ are distinguished as decision states.

At the Phase One beginning, state of $S$ is defined by vector $V_1$:

$$V_1 = \{o_1, o_2, \ldots, o_{1N}\} \quad (5)$$

An element $o_i(1 \leq i \leq N)$ contains the value $R_i^{in}$ of process $P_i$ and its signature $s_i$ over $R_i^{in}$. During Phase One process $P_i$ sends $o_i$ to every process $P_{Id}(1 \leq j \leq N, j \neq i)$ over $c_{ij}$ (1). Matrix $D1$ presents what was delivered by $S$ during Phase One. An element $d_{ij}^{1}$ is the value of $o_1$, as delivered over $c_{ij}$, i.e. where value of $c_{ij}$ is 0, indicating a faulty direct channel, the delivered value $d_{ij}^{1}$ is 0; otherwise $d_{ij}^{1} = o_i$:

$$D1 = \begin{bmatrix}
  d_{111} & d_{112} & \ldots & d_{11N} \\
  d_{121} & d_{122} & \ldots & d_{12N} \\
  \vdots & \vdots & \ddots & \vdots \\
  d_{1N1} & d_{1N2} & \ldots & d_{1NN}
\end{bmatrix} \quad (5.1)$$

On receiving $d_{ij}^{1} \neq 0$ from process $P_i$, process $P_j$ executes the system’s Hash function over $R_i^{in}$ component of $d_{ij}^{1}$, writes the outcome as $h_i$ component of element $e_i$ of its vector $E_j$, uses the signature $s_i$ component of $d_{ij}^{1}$ to verify the signature, writes $s_i$ as the $s_i$ component of element $e_i$ of its vector $E_j$, and in case of correctly verified signature writes $R_i^{in}$ in its vector $R^{Out}$ (4).

On completion of Phase One, every process $P_j$ has formed a vector of extended digital signatures $E_j$:

$$E_j = \{e_{1j}, e_{2j}, \ldots, e_{Nj}\} \quad (5.2)$$

Next, $P_j$ forms $v_{2j}$ containing the vector of extended digital signatures $E_j$ and a digital signature of $P_j$ over the content of vector $E_j$.

At the Phase Two beginning, state of $S$ is defined by vector $V_2$:

$$V_2 = \{v_{21}, v_{22}, \ldots, v_{2N}\} \quad (6)$$

During Phase Two each process $P_i$ sends $v_{2i}$ to every other process $P_j(1 \leq j \leq N, j \neq i)$ over $c_{ij}$. Matrix $D2$ presents the outcome of Phase Two. An element $d_{2ij}$ is $v_{2j}$ as delivered over $c_{ij}$:

$$D2 = \begin{bmatrix}
  d_{211} & d_{212} & \ldots & d_{21N} \\
  d_{221} & d_{222} & \ldots & d_{22N} \\
  \vdots & \vdots & \ddots & \vdots \\
  d_{2N1} & d_{2N2} & \ldots & d_{2NN}
\end{bmatrix} \quad (6.1)$$

In case of no process or link failures, all $d_{2ij}$ contain the same value. Process $P_j$ having received $(N - 1)$ vectors $d_{2ij}$ that have the same value as the value $d_{2ij}$ that $P_j$ received from itself, knows that every other process $P_i$ has received exactly the same values. So $P_j$ obtains the value upon which an agreement has been reached with execution of a pre-defined deterministic function that takes as input the content of vector $R_j^{Out}$.

### 3.3 Handling Faults and Asynchrony

The model assumes signed messages able to prove its authenticity to any process of the system. So, a Byzantine process may deceive about its value, but may not alter values received from other processes and send the altered values without revealing itself as malicious [47]. Thus the model limits the undetected behavior of Byzantine processes to just failure to send a protocol message to some or all processes or deliberately delay the sending [19].

### Handling Byzantine Deceit

During Phase Two the consensus algorithm exchanges extended signatures only. The model has to ensure that Byzantine consensus can still be solved with $N = 2F + 1$ processes. A Byzantine-faulty process $P_f$ may send different versions of $o_1, o_2, \ldots, o_{1N}$ to $P_j$ and $P_k(1 \leq j \leq N, 1 \leq k \leq N, j \neq k)$, thus deceiving about own value. If $R_i^{in}$ and $s_i$ do not match, $o_1$ is ignored; otherwise $P_j$ includes $e_i$ in vector $E_j$ and $P_k$ includes $e_i$ in vector $E_k$.

On completion of Phase Two, if vectors $E_j$ and $E_k$ distributed to all processes contain different versions of $e_i$, then $P_i$ reveals itself as Byzantine-faulty and all correct processes drop $R_i^{in}$ from the consensus vector $R^{Out}$. Sufficient is to verify that the different $h_i$ of each version is correctly signed with the different $s_i$ of its version. Hence system $S$ reveals the deceit and terminates consensus rounds with no need of more than $F + 1$ correct processes.

During Phase One: A Byzantine-faulty process $P_i$ may send more than one value as its $R_i^{in}$ value during Phase One. These values may be correctly or incorrectly signed. Process $P_i$ is supposed to send value $o_1$ to every process $P_j(1 \leq j \leq N, j \neq i)$ over $c_{ij}$. Yet
the value \( v_1 \) that process \( P_i \) sends is not the same to every \( P_j \):

\[
v_1 = \{ v^i_1 \}
\]

Process \( P_i \) may send different versions of \( v_1 \) to \( P_j (1 \leq j \leq N, j \neq i) \) and \( P_k (1 \leq k \leq N, k \neq i, k \neq j) \). Processes \( P_j \) and \( P_k \) execute Hash function over \( R_1^i \) to produce \( h \) and verify correctness of the received signature \( s_j \). If \( R_1^i \) and \( s_j \) do not match, \( v_1 \) is ignored; otherwise \( P_j \) includes \( h \) and \( s_j \) as element \( e_j \) in vector \( E_j \) and \( P_k \) includes \( h \) and \( s_j \) as element \( e_j \) in vector \( E_k \). During Phase Two, vectors \( E_j \) and \( E_k \) containing different versions of \( h \) will be distributed to all processes, thereby revealing \( P_i \) as Byzantine-faulty. So, all correct processes \( P_j (1 \leq j \leq N, j \neq i) \) will drop \( R_1^i \) from their vector \( R_{Out}^i \).

During Phase Two: Process \( P_i \) may deceive about value \( v_1 \) it has received from process \( P_j (1 \leq j \leq N, j \neq i) \) during Phase One. Value \( v_2 \) process \( P_i \) sends during Phase One contains \( E_i \) and signature of \( P_i \) over \( E_i \):

\[
E_i = \{ e_{1i} \ e_{2i} \ \ldots \ e_{Ni} \}
\]

Yet the value \( v_2 \) sent to at least one process \( P_k (1 \leq k \leq N, k \neq i) \) contains at least one value \( e_{ji} \) that does not reflect correctly the value \( v_1 \) that \( P_i \) has received from process \( P_j (1 \leq j \leq N, j \neq i, j \neq k) \). This deceit cannot be successful: a \( h \) incorrectly reflecting \( R_1^i \) has no chance to match the signature \( s_j \) of process \( P_j \).

Process \( P_i \) may try to deceive about the value received from process \( P_j (1 \leq k \leq N, k \neq i) \) by including a false \( s_j \) in its vector \( E_i \). However, \( P_i \) cannot produce a false signature \( s_k \) that matches with the false \( h \). Matching signature can be produced only by process \( P_k \). Whatever values \( h_k \) and \( s_k \) process \( P_i \) sends, the non-matching between these values will be easily detected by the recipient processes. So, process \( P_i \) cannot prevent consensus, but reveals itself as Byzantine-faulty.

**Handling Link Failure or Asynchrony**

Link failure is an arbitrary behavior of a one-way link \( c_{ij} \) between two processes \( P_i (1 \leq i \leq N) \) and \( P_j (1 \leq j \leq N, j \neq i) \) that prevents correct transmission of messages from the sender process \( P_i \) to the receiver process \( P_j \). Asynchrony of the one-way link \( c_{ij} \) is arbitrary behavior in regard to delivery time of the communication medium between processes \( P_i \) and \( P_j \). Link \( c_{ij} \) is represented in matrix \( C (1) \) with value 1 if it operates correctly and synchronously. It has value 0 if it does not deliver within the synchrony bounds.

The model does not distinguish faulty from asynchrony link and does not need to. It handles both with indirect delivery. It does not need the receiving end of a link to distinguish a faulty/asynchronous link from possible process failure on the sending end.

The model prescribes that in addition to the protocol rules, a correct process \( P_i (1 \leq i \leq N) \) must follow one more rule. On first receiving of a signed message \( m_{P_i,j} \), directly from the signatory \( P_j (1 \leq j \leq N, j \neq i) \) or indirectly, \( P_i \) multicasts \( m_{P_i,j} \) to every \( P_k (1 \leq k \leq N, k \neq i, k \neq j) \) with the signature of \( P_j \) over \( m_{P_i,j} \).

This rule ensures (where possible) that faulty or asynchrony links will be circumvented with 2-hop or 3-hop indirect delivery. It enacts attempts to deliver message \( m_{P_i,j} \) via different 2-hop indirect channels (2) and possible attempts to deliver \( m_{P_i,j} \) via different 3-hop indirect channels (3).

### Stop-Failed Process

A stop-failed process is one that executes correctly but stops at any time and once stopped, cannot restart. A stop-failed process \( P_i (1 \leq i \leq N) \) cannot send any \( v_1 \) or \( v_2 \) messages to any other process \( P_j (1 \leq j \leq N, j \neq i) \) over \( c_{ij} \) and cannot receive any \( v_1 \) or \( v_2 \) messages from any other process \( P_j \) over \( c_{ij} \). The effect of a stop-failed process \( P_i \) on the functioning of system \( S \) is the same as:

- Simultaneous failure/asynchrony of all links where \( P_i \) is sender, represented by the entire row with index \( i \) of matrix \( C (1) \), i.e. every element \( c_{ij} (1 \leq j \leq N) \) in matrix \( C \) having value 0; and
- Simultaneous failure/asynchrony of all links where \( P_i \) is receiver, represented by the entire column with index \( i \) of matrix \( C (1) \), i.e. every element \( c_{ij} (1 \leq j \leq N) \) in matrix \( C \) having value 0.

**Note:** Process failure presented as link failures provides a common denominator for assessment of the combined effect of simultaneous process failures and link failures/asynchrony and the effect of processes’ possible Byzantine behavior in its variety and scale.

The model does not distinguish whether a missing message is caused by a process stop-failure or anything else. Yet, it takes into consideration that a stop-failed process spoils all 2-hop and 3-hop indirect channels that depend on its active participation. Stop-failed sender during multicast has the effect of Byzantine omission.

System \( S \) needs just \( F+1 \) correct processes to be able terminate consensus in spite of \( F \) stop-failed processes, as a consequence of the eliminated chance for successful Byzantine deceit [47] [19]. With this model, correctness of the above statement is subject to additional condition: As long as the \( F \) stop-failed processes do not invalidate all indirect channels allowing a correct process to have two-way communication with the rest of the \( F \) correct processes.

**Handling Byzantine Omission**

Every process \( P_i (1 \leq i \leq N) \) is required by consensus protocol to send each protocol message to every other process \( P_j (1 \leq j \leq N, j \neq i) \) over \( c_{ij} \). A Byzantine-faulty \( P_i \) may omit sending to \( P_j \). Omission by \( P_i \) to send to \( P_j \) over \( c_{ij} \) has exactly the same effect on functioning of system \( S \) as the event of failure of link \( c_{ij} \) and is handled in the same way.

Process \( P_i \) can omit sending to any process \( P_j \) without being detected. Sending to at least one process of a group of \( F+1 \) correct processes with two-way channels between themselves allows the model to tolerate its omissions. Omitted sending to all of the group of \( F+1 \) correct processes has the effect of a stop-failed process.

**Handling Byzantine Delay**

Every process \( P_i (1 \leq i \leq N) \) is required by consensus protocol to send each protocol message to every process \( P_j (1 \leq j \leq N, j \neq i) \) over \( c_{ij} \) with no delay. A delayed sending by \( P_i \) to a process \( P_j \) has the same effect on functioning of consensus system \( S \) as sending over asynchronous link, handled exactly as a Byzantine omission.

### 3.4 Correctness of Consensus

Consensus requires existence of a group \( G \), which comprises \( G \geq F+1 \) correct processes communicating via direct or indirect channels with bounded delivery.

**Correctness Conditions**

**Agreement:** Every two processes \( P_i \) and \( P_j \) from \( G \) compute the same value for their output registers \( R_{Out}^i \) and \( R_{Out}^j \).

**Termination:** All processes in \( G \) decide within 4 RTTB intervals.
4 THE ALGORITHM

4.1 Synchronization

The system model assumes synchronized processes [30, 36, 18, 42, 53], with max lag bounded to less than half RTTB (round trip transmission bound). Sufficient condition for ensuring the assumed max lag is every process to have access to a Time server with embedded GPS receiver utilizing atomic-clock synchronization systems [14, 5, 41]. The processes of a system with synchronized clocks need only to fix the start time of the first consensus round and then periodically re-synchronize the clocks. The last process joining the system broadcasts a signed message setting the start time. Every recipient of that message, on first direct or indirect receiving also broadcasts it together with its original signature.

4.2 Flexible Time-bound

Public networks like the Internet operate with fluctuating hourly workload over some backbone elements, causing large fluctuations of the normal delivery time. Hence, flexible time-bound for delivery is a precondition for efficiency in ensuring synchrony. The issue is not essential for this paper, so we only acknowledge it briefly.

Backbone routers’ congestion control algorithms use random early detection [32, 24, 34] to prevent buffers overflow by reducing transmission speed of individual flows, possibly multiple times below the maximum. This way, Transmission Control Protocol (TCP) allows routers to handle surges of traffic and avoid channels partitioning. Under partially synchronous communication and the consensus model of eventual synchrony, execution might halt as a result of a randomly delayed synchronous flow, seen as asynchrony.

One way of dealing with such random asynchrony is to switch to a larger time-bound for message delivery. Its outcome would be similar to that of routers operating with a fraction of their maximum throughput, yet being unable to eliminate the possibility for random asynchrony. More efficient is a feature similar to TCP congestion throughput, yet being unable to eliminate the possibility for random asynchrony. More efficient is a feature similar to TCP congestion control.

4.3 Handling Faulty/Asynchronous Links

The algorithm operates in a deterministic data-driven manner by responding to events from a pair of event categories in the sequence of appearing: expiry of time intervals and receiving peer messages.

Listing 1: Processing Timeout Events in C++

```cpp
void OnTimeout( BYTE Event )
{
    switch ( Event )
    {
        case START1 : // Start Phase One
            BroadcastMssgPhaseOne() ;
            break ;
        case START2 : // Start Phase Two
            BroadcastMssgPhaseTwo() ;
            break ;
    }
}
```

Listing 1 presents the algorithm’s part that maintains synchronization between consensus rounds of process $P_i (1 \leq i \leq N)$ in system $\mathcal{S} (N \geq 3, N \geq 2F + 1)$ with the rest of processes $P_j (1 \leq j \leq N, j \neq i)$. Function OnTimeout() responds to 2 time events:

- START1 prompts process $P_i$ to broadcast its own MSGPHASE1 message to all processes $P_j (1 \leq j \leq N, j \neq i)$.
- START2 prompts process $P_i$ to broadcast its own MSGPHASE2 message to all processes $P_j (1 \leq j \leq N, j \neq i)$.

Listing 2: Processing Received Messages in C++

```cpp
void OnRcvdMssg( Mssg* pMSSG )
{
    switch ( pMSSG->m_Mssg_Type )
    {
        case MSGPHASE1 : // Rcvd a Direct Mssg Phase One
            if ( HasMssgBeenRcvd ( pMSSG ) ) break;
            ProcessMssgPhaseOne( pMSSG ) ;
            RetrnsmMssgPhaseOne ( pMSSG ) ;
            break;
        case MSGPHASE2 : // Rcvd a Direct Mssg Phase Two
            if ( HasMssgBeenRcvd ( pMSSG ) ) break;
            ProcessMssgPhaseTwo( pMSSG ) ;
            RetrnsmMssgPhaseTwo ( pMSSG ) ;
            break;
    }
}
```

Listing 2 presents the algorithm’s part that handles and responds to the messages that process $P_i (1 \leq i \leq N)$ receives from the rest of processes $P_j (1 \leq j \leq N, j \neq i)$. Function OnRcvdMssG() responds to 4 types of messages:

- MSGPHASE1 is a message that process $P_i (1 \leq i \leq N)$ receives directly from every process $P_j (1 \leq j \leq N, j \neq i)$. On first receiving, directly or via RTRNSMTD1, $P_i$ verifies the signature, records the message in its internal storage, and multicasts it at RTRNSMTD1 message to every process $P_k (1 \leq k \leq N, k \neq i, k \neq j)$.
- RTRNSMTD1 is a message signed by $P_j (1 \leq j \leq N, j \neq i)$ and received by $P_i$ sent by $P_k (1 \leq k \leq N, k \neq i, k \neq j)$ in response to MSGPHASE1 from $P_j$.
- MSGPHASE2 is a message that process $P_i (1 \leq i \leq N)$ receives directly from every process $P_j (1 \leq j \leq N, j \neq i)$. On first receiving, directly or via RTRNSMTD2, $P_i$ verifies the signature, records the message in its internal storage, and multicasts it as RTRNSMTD2 message to every process $P_k (1 \leq k \leq N, k \neq i, k \neq j)$.
- RTRNSMTD2 is a message signed by $P_j (1 \leq j \leq N, j \neq i)$ and received by $P_i$ sent by $P_k (1 \leq k \leq N, k \neq i, k \neq j)$ in response to MSGPHASE2 from $P_j$.

4.4 Algorithm Illustration

Figure 1 is a diagram that illustrates the algorithm operations in a worst-case scenario. On the diagram, all processes are correct and all presented links are synchronous. The model assumes that the processes start a round with max time lag bound to less than 1/2 RTTB interval. In the presented worst-case scenario:

- delivery of messages over the presented links takes the max allowed time of 1/2 RTTB;
- process $Q$ starts a consensus round after process $P$, lagging with the highest tolerated lag of nearly 1/2 RTTB interval;
5 PROOFS

5.1 Synchronous Consensus

We will show that under the model assumptions the algorithm operates with synchrony, as illustrated on Figure 1 where processes \(P, P', P'', Q, Q',\) and \(Q''\) complete their individual consensus rounds before expiry of 4 RTTB on their individual clocks.

**Theorem 1:** Synchronous consensus during incomplete synchrony. 
A system of synchronized (less than 1/2 RTTB lag) synchronous processes operating with a 3-hop delivery algorithm, where in a group \(G\) of \(F + 1\) correct processes every process has a synchronous link (1/2 RTTB delivery bound) or an indirect channel over synchronous links to and from every other process in \(G\), terminates within 4 RTTB intervals regardless of possibility all other links to be asynchronous.

**Proof**
The trivial case is when \(F + 1\) correct processes are directly connected to each other via \((F + 1) + F\) synchronous links. The proof must show that with a fraction of these links, allowing every pair of processes a direct or indirect bidirectional communication, in any combination of direct and 2-hop and 3-hop indirect channels, each of the \(F + 1\) correct process terminates within 4 RTTB intervals. 2-hop indirect channels take up to 1 RTTB interval to deliver a message. 3-hop indirect channels take up to 1.5 RTTB interval to deliver a message. In the worst-case scenario, the first process that starts a round receives a message from every other process before expiry of 2 RTTM on its clock, as 1/2 RTTB lag + 1.5 RTTB delivery + 2 RTTM. All other processes receive a message from every other process within a shorter time on their individual clocks. Hence, everyone of the \(F + 1\) correct processes in \(G\) receives the messages necessary and sufficient for termination within 4 RTTB intervals.

5.2 Highest Tolerance to Asynchronous Links

Gossiping (epidemic dissemination) is typically used to ensure message delivery under the weakest needs of correct links [9] [22] [29]. Our algorithm implements gossiping for tolerance to faulty and/or asynchronous links. It does so with up to 3-hop indirect delivery. Any additional hop increases the tolerance but delays termination.

**Lemma 1:** Epidemic dissemination of messages.
On existence of a path between sender and recipient of a message, an algorithm for epidemic dissemination will deliver the message.

**Proof:** In an epidemic algorithm every process of a system forwards the received messages over its output links. Thus, the algorithm utilizes all existing in the system multi-hop delivery routes.

**Lemma 2:** Consensus with epidemic dissemination of messages.
A synchronous consensus algorithm implementing epidemic dissemination of messages ensures highest tolerance to faulty links.

**Proof:** It follows from Lemma 1.

**Theorem 2:** Synchronous consensus with tolerance to asynchrony.
Epidemic dissemination of messages lets synchronous consensus algorithm terminate under highest number of asynchronous links.

**Proof**
Consider a synchronous consensus algorithm with epidemic dissemination and a system with 1. one-way links, where either only faulty or only asynchronous links prevent delivery. Both cases are handled in a way taking longer yet bounded time. In the case with only faulty links, let the smallest sufficient for consensus set of...
messages $M$ is delivered over one of the smallest sets of correct links $S$ allowing delivery [Lemma 1], i.e. with highest tolerated faulty links $F$ [Lemma 2], $F = L - S$. In the case with only asynchronous links, set $S$ delivers the same set $M$, thus tolerating $A$ asynchronous links, where $A = L - S$ hence $A = F$. As synchronous consensus is not possible with a larger $F$, it is also not possible with a larger $A$.

5.3 Leaderless Consensus

Theorem 3: Leaderless consensus over a medium susceptible to asynchrony and faults.

Consensus algorithm, of a system of synchronous processes that communicate over a medium susceptible to asynchrony and faults, does not need to operate with a leader if it can ensure direct or indirect bounded delivery of messages among the required for consensus majority of correct processes.

Proof

Let us assume existence of a group $G$ of correct processes, equal or larger than the required for consensus quorum, with ensured bounded delivery, so that every message received by one process of $G$ is also received by every process of $G$ and included a group $M$ messages individually maintained by every process in $G$. Not included in $M$ are the messages: 1) directly received after expiry of 1 RTTB on processor’s clock since the start of the current phase; 2) received over a 2-hop channel after expiry of 1.5 RTTB; and 3) received over a 3-hop channel after expiry of 2 RTTB intervals.

Thus, all processes in $G$ receive within a bounded time the same set of messages $M$, extract exactly the same data, and produce atomically consistent individual states. The consensus value computed by the processes of $G$ is produced from atomically consistent states, and used as input to instances of the same deterministic function. Hence the consensus values individually produced by the processes of $G$ are atomically consistent, as if proposed by a leader.

5.4 Ensured Both Safety and Bounded Liveness

Theorem 4: Possibility for deterministic Byzantine consensus with conceptually ensured simultaneous safety and bounded liveness in partial synchrony.

A synchronous deterministic Byzantine consensus algorithm, capable to tolerate asynchrony and faults on its communication medium, operates with conceptually ensured simultaneous safety and bounded liveness in the area within its boundaries of tolerance.

Proof

Use of asymmetrically authenticated messages [48] lets tolerate some asynchronous / faulty links. It prevents a Byzantine process from affecting consensus safety [see Handling Byzantine Deceit in subsection 3.3 and Theorem 3 proof]. Theorem 1 proves bounded delivery within the boundaries of tolerance to links’ asynchrony and/or faults and faulty processes [see Handling Byzantine Omission and Handling Byzantine Delay in subsection 3.3]. Theorem 3 proves that under bounded delivery consensus does not need leader. Having ensured leaderless consensus and bounded delivery in spite of links’ asynchrony and/or faults and in spite of processes’ crashes and/or omissions and/or delays, nothing capable to prevent bounded termination has left non-eliminated. Hence, within its boundaries of tolerance the algorithm operates with conceptually ensured simultaneous safety and bounded liveness.

6 SUMMARY OF THE RESULTS

This work demonstrated the following new features and qualities related to deterministic consensus algorithms in partial synchrony:

1. A concept for proactive handling of asynchrony with virtual indirect multi-hop channels.
2. Higher tolerance to links asynchrony than the leader-based algorithms with asymmetrically authenticated messages.
3. Tolerance to any fault of any process, within the tolerated limits of faults and asynchrony, with no effect on bounded liveness.
4. A fully decentralized deterministic Byzantine consensus algorithm with precisely defined tolerance to asynchronous links.
5. Conceptually ensured bounded termination and conceptually ensured simultaneous safety and bounded liveness.
6. Simulator of consensus work, revealing events’ permutations capable to prevent bounded liveness [see the Appendix].
7. A methodology for regression analysis of simulation results and extrapolation [see the Appendix].
8. Analytically defined tolerance to asynchrony in consensus systems per number of faulty processes [see the Appendix].
9. Superlinear increase of the tolerated asynchronous links with every added pair of system processes [see the Appendix].

7 CONCLUSION

We presented a synchronous deterministic Byzantine consensus algorithm in partial synchrony. It operates with the weakest needs for communication synchrony, has no single point of failure, and ensures simultaneous safety and bounded liveness

These features are outcome from a trade-off: the tolerance to faults and asynchrony is paid with limits on the efficient scalability and halved throughput. The gained non-existing at present features enable creation of new technologies with impact in areas ranging from financial services to cyber resilience.

The algorithm lets build technologies solving the obstructed efficiency of financial markets: in terms of time – with real-time settlement of trades; and in terms of consistency – with elimination of conditions allowing creation of phantom stocks.

A decentralized infrastructure built with the algorithm operates with enforced quantifiable cyber resilience. Multiple concurrent security breaches installing malicious code and multiple faulty and/or asynchronous links within the tolerated limits cannot disrupt it.

ACKNOWLEDGMENTS

This work is partly sponsored by the Australian Federal Government through the Research and Development Tax Incentive Scheme. The author expresses his gratitude to Simeon Simoff for his help with combinatorics, suggestions on formalization, and comments.
APPENDIX
A BOUNDARIES OF TOLERANCE

Note: Throughout the next two sections, term 'correct link' denotes 'correct synchronous link' and term 'faulty link' denotes 'faulty or asynchronous link' for neatness of the presentation.

Theorem 1 demonstrated that circumvention of faulty links with 3-hop indirect delivery is paid with an increase of time needed to establish agreement from 2 to 4 RTTB intervals. Paying that price requires knowledge of what exactly will be received in return. So, we need to find out the number of tolerated faulty links per 0, 1, 2, ..., F faulty processes for every system of 2F + 1 processes.

To formulate the goal, for a particular number of total processes we need to compute all permutations of faulty links and faulty processes per each tolerated number of faulty processes and test each computed permutation whether each process in a group of F+1 correct processes receives a message from every process in that group. The goal is to discover the highest number of faulty links per faulty processes where every permutation solves consensus.

A.1 The Simulator

Different permutations of the same number of correct links may need 2-hop or 3-hop indirect channels to solve consensus. With a system of 7 processes and a few faulty, the possible permutations with faulty processes and faulty links are millions or billions.

Discovering the highest number of tolerated faulty links per faulty processes with solvable consensus requires a simulator to compute all permutations and test each one individually. The computation could happen on a single computer. A one-way link could be implemented as a shared memory, which can be modified by exactly one process and read by exactly one other process.

With this objective, we built a simulator that distributes the algorithm’s messages in a fully sequential manner, according to the permutation of correct processes and correct links. A completed distribution of messages is considered a completion of consensus round. Receiving a message by a recipient from a sender is considered receiving of both Phase One and Phase Two messages.

Simulation was implemented according to the system model’s common denominator for process faults, links asynchrony, and link faults. Omissions and/or delays of a Byzantine-faulty process are simulated in their worst-case scenario, where the max harm of a Byzantine-faulty process is exactly the same as with a stop-failed process. So, any form of faulty process is modeled as a stop-failed process. Asynchronous link is modeled as a faulty link, i.e. with value 0 in matrix C (1). A stop-failed process is modeled with value 0 of anyone of its inbound and outbound links in matrix C (1).

On completed distribution of messages, the simulator checks for the existence of a group of F + 1 correct processes, wherein every process of the group has received a message from every other process of that group. If yes, the consensus is considered solved.

The simulator operates in the following manner:

Input: As input it takes: a) total number of processes; b) number of correct processes; and c) number of correct links.

Execution: It creates objects representing processes and links, traces all permutations of correct processes and correct links, and checks each permutation for ability to solve consensus.

A.2 Computation with a 5-process System

According to the model, a 5-process system can tolerate up to 2 faulty (Byzantine or stop-failed) processes. We performed three series of computations, each comprising a series of 21 computations itself – the first with 0 faulty links and the last with 20. Every computation traces all permutations of faulty processes and faulty links and tests each permutation for ability to solve consensus.

Series with Two Faulty Processes

As it is shown on Figure 2, a 5-process system operating with 2 faulty processes ensures consensus with up to 1 faulty link. In presence of 2 faulty links, the system solves consensus with 1,840 permutations out of 1,900 total, hence operates with 0.9968 probability to solve it.

Series with One Faulty Process

As Figure 3 shows, a 5-process system operating with 1 faulty process ensures consensus with up to 3 faulty links. In presence of 4 faulty links, the system solves consensus with 24,195 permutations out of 24,225 total, hence operates with 0.9987 probability to solve consensus.
In presence of 5 faulty links, the system solves consensus with 24,074 permutations out of 24,225 total, hence operates with 0.9938 probability to solve it.

**Series with No Faulty Process**

Figure 4 as well depicts the diagrams presenting: total permutations, permutations that solve consensus, and probability to solve consensus, per number of faulty links. In regard to probability, compared to the diagrams on the previous two figures, the major change again is the expansion of the region with ensured solving of consensus. Yet with no faulty processes the expansion is huge: from tolerance of 3 faulty links with 1 faulty process to tolerance of 9 faulty links.

As Figure 4 shows, a 5-process system operating with no faulty process ensures consensus with up to 9 faulty links. In presence of 10 faulty links, the system solves consensus with 184,696 permutations out of 184,756 total, hence operates with 0.9986 probability to solve it. In presence of 11 faulty links, the system solves consensus with 167,420 permutations out of 167,960 total, hence operates with 0.9967 probability to solve it.

**B.1 Solution Idea**

With a few exceptions, tolerance boundaries of a system of any size cannot be established with a mainstream computing technology. The alternative is to do it analytically in the following manner:

- Extract as much data as possible with computation;
- Use the computation data to fill a regression table;
- Use the regression table to build tolerance equations;
- Apply the equations to a systems of any size.

Computations with a 6-process system with no faulty processes show that the system tolerates up to 4 faulty links. This means that under any permutation of 4 faulty links, the system unconditionally delivers all sent messages. The same system increases its tolerance to faulty links from 4 to 7 when it has to deliver messages of any 5 processes to the same 5 processes, and from 7 to 15 faulty links when it has to deliver from any 4 processes to the same 4 processes.

**Creation of regression table**

First step is to fill a row with the computed results showing how the tolerance to faulty links in delivery from $N$ to $N$ processes in an $N$-process system increases with an increment of $N$ with 1. Then to fill the next row with experimentally computed results showing how the tolerance to faulty links in delivery from $(N - 1)$ to $(N - 1)$ processes in an $N$-process system increases with an increment of $N$ with 1, then from $(N - 2)$ to $(N - 2)$ processes, etc.

**Illustration of the approach**

Consider the tolerance to faulty links $f$ of the 7-process system on Figure 5 and the possible regression. The system tolerates up to $F = 3$ faulty processes and number of $f$ faulty links, where with:

- $F = 3$ faulty processes, it is practically a 4-process system, which tolerates $f = 3$ in delivering across the 4 correct processes.
- $F = 2$ faulty processes, it is practically a 5-process system, which tolerates $f = 3$ in delivering across the 5 correct processes, and tolerates additional $4f = 2$ faulty links in delivering across any 4 correct processes, so the resulting tolerance is $f = 5$ faulty links.
- $F = 1$ faulty processes, it is practically a 6-process system, whose tolerance is equal to sum of tolerances in delivering across the 6 correct processes and the additional tolerance in delivering across any 5 correct processes and across any 4 correct processes.
B.2 Regression Table From Computations

Table 1 presents the results of computations on multiple simulator systems within a time limit set to 2 months per computation series. Individual computation series were performed with varied total number of processes $N = 3, 4, 5, ..., 9$. Each one established tolerance to faulty links $f$ in delivery of $N$ messages across $N$ processes, $(N-1)$ messages across $(N-1)$ processes, etc.

Table data allows observation of the following 4 patterns:

**Pattern 1.** In delivery of $N$ messages across $N$ processes, an increment of $N$ with $1$ increases the faulty link tolerance $f$ with $1$.

**Pattern 2.** In delivery of $(N-1)$ messages across $(N-1)$ processes, starting from $N = 4$, an increment of $N$ with $1$ increments $\Delta$ with $1$. The meaning of $\Delta$ is the additional tolerance $(+f)$, caused by simplifying the objective by $1$, i.e. from delivery of $N$ messages across $N$ processes to delivery of $(N-1)$ across $(N-1)$ processes.

**Pattern 3.** In delivery of $(N-2)$ messages across $(N-2)$ processes, starting from $N = 6$ there is a repetition of Pattern 2. Hence, Pattern 3 can be formulated as: Starting from $N = 4$, an increment of $N$ with $2$ causes a new start of Pattern 2 in regard to a simplified by $1$ objective, i.e. from delivery of $(N-1)$ messages across $(N-1)$ processes to delivery of $(N-2)$ messages across $(N-2)$ processes.

**Pattern 4.** With $N - f = F + 1$, where $F$ is the highest tolerated number of faulty processes, starting from $N = 3$, an increment of $N$ with $2$ causes an increment of $\Delta$ with $2$. This pattern is clearly observed in the following sequence: 1) $N = 3$, delivery from $(N-1)$ to $(N-1)$, $\Delta = 3$; 2) $N = 5$, delivery from $(N-2)$ to $(N-2)$, $\Delta = 4$; and 3) $N = 7$, delivery from $(N-3)$ to $(N-3)$, $\Delta = 6$.

### Table 1: Computation Results

| Delivery Type | $N$ | $3$ | $4$ | $5$ | $6$ | $7$ | $8$ | $9$ |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|
| $N$ to $N$    | $f$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ | $7$ |
| $(N-1)$ to $(N-1)$ | $\Delta$ | $2$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ |
| $(N-1)$ to $(N-1)$ | $f$ | $3$ | $3$ | $5$ | $7$ | $9$ | $11$ |
| $(N-2)$ to $(N-2)$ | $\Delta$ | $5$ | $4$ | $1$ | $2$ |
| $(N-2)$ to $(N-2)$ | $f$ | $8$ | $9$ | $8$ | $11$ |
| $(N-3)$ to $(N-3)$ | $\Delta$ | $7$ | $6$ |
| $(N-3)$ to $(N-3)$ | $f$ | $15$ | $17$ |

**B.3 Regression Table Analytically Expanded**

Table 2 presents together the results from computation and from analytical extrapolation, based on the observed cause and effect patterns between the incremented number of system processes $N$ and the various $\Delta$s. Recall that value of $f$ in a particular row and column is computed by adding the value of $\Delta$ in the previous row of the same column to the of preceding $f$ in that column.

### Table 2: Computation and Extrapolation Results

| Delivery Type | $N$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ | $7$ | $8$ | $9$ | $10$ | $11$ |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| $(N-1)$ to $(N-1)$ | $\Delta$ | $2$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ | $7$ | $8$ |
| $(N-1)$ to $(N-1)$ | $f$ | $3$ | $3$ | $5$ | $7$ | $9$ | $11$ | $13$ | $15$ | $17$ |
| $(N-2)$ to $(N-2)$ | $\Delta$ | $5$ | $4$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ |
| $(N-2)$ to $(N-2)$ | $f$ | $8$ | $9$ | $8$ | $11$ | $14$ | $17$ | $20$ | $23$ |
| $(N-3)$ to $(N-3)$ | $\Delta$ | $7$ | $6$ | $1$ | $2$ | $3$ | $4$ |
| $(N-3)$ to $(N-3)$ | $f$ | $15$ | $17$ | $15$ | $19$ | $23$ | $27$ |
| $(N-4)$ to $(N-4)$ | $\Delta$ | $9$ | $8$ | $1$ | $2$ |
| $(N-4)$ to $(N-4)$ | $f$ | $24$ | $27$ | $24$ | $29$ |
| $(N-5)$ to $(N-5)$ | $\Delta$ | $11$ | $10$ |
| $(N-5)$ to $(N-5)$ | $f$ | $35$ | $39$ |

**B.4 Examples: Use of Extrapolation**

Tolerance of a 9-process system to faulty links per actual number of faulty processes $F$ can be computed (with $F = 4, F = 3$, and $F = 2$) using data from Table 1 and (with $F = 3$ and $F = 0$) using data from Table 2 in the following manner:

- With $F = 4$, a 9-process system is equivalent to a 5-process system, which tolerates $f = 3$ in delivering 5 messages to 5 processes (Table 1).
- With $F = 3$, the system is equivalent to a 6-process system, which tolerates $f = 4$ in delivering 6 messages to 6 processes (Table 1), and tolerates additional $+f = 3$ in delivering 5 message to 5 processes (Table 1), so the resulting tolerance is $f = 7$.
- With $F = 2$, the system is equivalent to a 7-process system, which tolerates $f = 5$ in delivering 7 messages to 7 process (Table 1), tolerates additional $+f = 4$ in delivering 6 messages to 6 processes (Table 1), and tolerates $+f = 2$ in delivering 5 messages to 5 processes (Table 1), so the resulting tolerance is $f = 11$.
- With $F = 1$, the system is equivalent to a 8-process system, which tolerates $f = 6$ in delivering 8 messages to 8 processes (Table 1), tolerates additional $+f = 5$ in delivering 7 messages to 7 processes, tolerates additional $+f = 3$ in delivering 6 messages to 6 processes (Table 2, Pattern 3), and tolerates additional $+f = 1$ in delivering 5 messages to 5 processes (Table 2, Pattern 3), so the resulting tolerance is $f = 15$.
- With $F = 0$, the system is a 9-process system, which tolerates $f = 7$ in delivering 9 messages to 9 processes (Table 1), tolerates additional $+f = 6$ in delivering 8 messages to 8 processes (Table 2, Pattern 3), additional $+f = 4$ in delivering 7 messages to 7 processes, additional $+f = 2$ in delivering 6 messages to 6 processes (Table 2, Pattern 3), and additional $+f = 8$ in delivering 5 messages to 5 processes (Table 2, Pattern 4), so the resulting tolerance is $f = 27$.

Tolerance of an 8-process system to faulty links per actual number of faulty processes $F$ can be computed (with $F = 3, F = 2,$ and
Figure 6: Tolerance to Async/Faults (Computed + Analytical)

$F \in \{1\}$) using data from Table 1 and (with $F = 0$) using data from Table 2 in the following manner:

- With $F = 3$, an 8-process system is equivalent to a 5-process system, which tolerates $f = 3$ in delivering 5 messages to 5 processes (Table 1).
- With $F = 4$, the system is equivalent to a 6-process system, which tolerates $f = 4$ in delivering 6 messages to 6 processes (Table 1), and tolerates additional $+f = 3$ in delivering 5 message to 5 processes (Table 1), so the resulting tolerance is $f = 7$.
- With $F = 5$, the system is equivalent to a 7-process system, which tolerates $f = 5$ in delivering 7 messages to 7 processes (Table 1), tolerates additional $+f = 4$ in delivering 6 messages to 6 processes (Table 1), and tolerates $+f = 2$ in delivering 5 messages to 5 processes (Table 1), so the resulting tolerance is $f = 11$.
- With $F = 0$, it is an 8-process system, which tolerates $f = 6$ in delivering 8 messages to 8 processes (Table 1), tolerates additional $+f = 5$ in delivering 7 messages to 7 processes (Table 1), tolerates additional $+f = 3$ in delivering 6 messages to 6 processes (Table 2, Pattern 3), and tolerates additional $+f = 1$ in delivering 5 messages to 5 processes (Table 2, Pattern 3), so the resulting tolerance is $f = 15$.

Note: It might appear strange that an 8-process system with no faulty processes tolerates $f = 15$, while a 7-process system with no faulty processes tolerates $f = 17$. This outcome illustrates why an additional process can make achieving an agreement harder, yet with no contribution to the tolerance to link faults or asynchrony. Consensus of an 8-process system requires delivery of 5 messages across 5 correct processes, while consensus of a 7-process system requires delivery of 4 messages across 4 correct processes.

**B.5 Tolerance Curves with Extrapolation**

Figure 6 depicts a family of curves, where a curve represents the tolerance of a consensus system to faulty and/or asynchronous links per number of tolerated faulty processes. Tolerance of a 5-process system, a 7-process system, and a 9-process system with 4 and with 3 faulty processes is computed entirely with the simulator.

Tolerance of a 9-process system with 2 faulty processes, with 1 faulty process, and with no faulty process is computed analytically with extrapolation. Tolerance of an 11-process system is computed entirely analytically with extrapolation.

**B.6 Conditions for Bounded Termination**

The following equations present tolerance of consensus system to the sum of faulty and asynchronous links $f$ as function of system connectivity $c$ (Graph theory meaning of term connectivity [15]) and the number of faulty processes $F_t$ at a point of time $t$:

- With odd total number of processes and no faulty processes:
  
  $$f < c + c/2 + (c/2)^2 \quad (8.1)$$

- With odd total number of processes and $F_t$ faulty processes:
  
  $$f_t < c/2 + (c/2)^2 - F_t \times c/2 \quad (8.2)$$

- With even total number of processes and no faulty processes:
  
  $$f < (c + 1)/2 + ((c + 1)/2)^2 - (c + 1)/2 \quad (8.3)$$

- With even total number of processes and $F_t$ faulty processes:
  
  $$f_t < (c + 1)/2 + ((c + 1)/2)^2 - (F_t + 1)(c + 1)/2 \quad (8.4)$$

These equations are the conditions for bounded termination.

**B.7 Final Comments**

All results presented in sections B and C are obtained with a previous version of the algorithm, where 3-hop delivery between processes $P$ and $Q$ requires: 1) $P$ to have a correct synchronous outbound link to a process $P'$ and a correct synchronous inbound link from the same process $P'$; 2) $Q$ to have a correct synchronous outbound link to a process $Q'$ and a correct synchronous inbound link from the same process $Q'$; and 3) $P'$ and $Q'$ to be part of the correct majority and have correct synchronous links between themselves.

The current version relaxes the 3-hop requirements. As Figure 1 shows, delivery between processes $P$ and $Q$ requires: 1) $P$ to have a correct synchronous link to a process $P'$ and a correct synchronous link from a process $P''$, where $P'$ and $P''$ can be but not necessarily are the same process; 2) $Q$ to have a correct synchronous link to a process $Q'$ and a correct synchronous link from a process $Q''$, where $Q'$ and $Q''$ can be but not necessarily are the same process; and 3) $P'$, $P''$, $Q'$, and $Q''$ to be part of the correct majority and have correct synchronous links between themselves.

Hence the current version’s tolerance to asynchronous and/or faulty links per number of faulty processes is higher than the presented here. Yet the actual borderline tolerance data is sensitive and kept confidential to prevent its use for fine tuning of coordinated cyber attacks on liveness of consensus networks that operate with the presented algorithm.