The Consensus Machine: Formalising Consensus in the Presence of Malign Agents

A. W. Roscoe[1,2,3(✉)], Pedro Antonino[1], and Jonathan Lawrence[1]

[1] The Blockhouse Technology Ltd., Oxford, UK
{pedro,jonathan}@tbtl.com
[2] Department of Computer Science, University of Oxford, Oxford, UK
awroscoe@gmail.com
[3] University College Oxford Blockchain Research Centre, Oxford, UK

Abstract. This paper is on the application of formal modelling in CSP and associated verification to decision making in decentralised systems. In particular we look at the problem of ensuring that decentralisation cannot allow two separate and apparently valid decisions to arise when exactly one is required. This is motivated by an approach to blockchain consensus where a primary choice mechanism may need to be supplemented by a back-up that comes into action if the primary one is seemingly blocked.

Keywords: Consensus · State machine · Blockchain · Process algebra · Formal methods

Dedication to He Jifeng on the occasion of his 80th birthday:

Jifeng and I worked together for many years at Oxford developing theories of verification and making them usable. Indeed we have gone on to use them successfully in many contexts, always rooted in algebra and abstraction. In this paper we show how these same two ideas can improve understanding in a relatively new domain—blockchain.
Bill Roscoe

1 Introduction

Consensus is a classical problem in the area of distributed, decentralised systems. It has regained the attention of the scientific community with the advent of blockchains. In these, consensus is used to ensure that participants in this distributed system agree what is the next block in this ever-extending chain of blocks; an initial *genesis block* is initially agreed amongst participants. This agreement on the next block in the chain is usually referred to as the *finality problem*, namely, how to determine the next (agreed-upon) *final* block. Here,

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 136–162, 2023.
https://doi.org/10.1007/978-3-031-40436-8_6

final means that it is immutable and will have that position in the blockchain for ever; it does not mean that it is the last block in the chain. A block in such a chain represents a sequence of transactions, each of which causes the state of the blockchain to evolve. So, in broad terms, blockchains are transaction-processing systems possessing an integrity-protected transaction history.

In this paper, we propose the concept of a *hierarchical consensus machine* as a means to solve the finality problem efficiently. In our formulation, this machine is composed of two consensus machines, say G and H, each of which is implemented by a distributed collection of agents, and it decides on a value that is agreed upon by (most of) its (well-behaved) agents. The number agreeing will have reached some pre-agreed threshold. While G is designed to be safe but not live, H is both safe and live. Broadly speaking, safety means that the machine decides on a single correct value, whereas liveness means that the machine eventually comes to a decision. Our definition of *correct* here is that it is the conclusion of one or more good agents who always follow the rules.

As part of our hierarchical machine, we also propose a handover protocol that transfers control from G to H whenever G is unable to come to a (timely) decision; this protocol ensures the liveness of the hierarchical machine as a whole. We create this concept by first introducing a didactic account of consensus machines using a *unitary consensus machine*. Furthermore, we propose a type of *stochastic reasoning* that is a useful mathematical tool to establish bounds on the number of participants in the agreement to achieve safety and liveness. In fact, the reason for having a machine G that is only safe as part of our hierarchical machine is that we can demonstrate with this type of stochastic reasoning that a much smaller number of participants are required to achieve safety alone as compared to machines that are both safe and live. The smaller number of participants should allow for a more efficient consensus protocol. We count on the incentive structures of blockchains and on the fact that ultimately H will reach a consensus to motivate malign agents to cooperate with good agents to reach a consensus via G. Malign agents' misbehaviour is the reason why G is not dependably live. Thus, persuading them to behave in a collaborative way should make G (efficiently) come to a decision more often. We formalise our notion of a hierarchical consensus machine using the CSP process algebra. The notions that we present here could be adapted to the general problem of consensus provided they are used in a similar context. In this paper we concentrate solely on this binary G/H case, but evidently this is open to extension.

As blockchains are being adopted by many industry sectors, finding efficient consensus protocols in this context has become a relevant research challenge. Our *hierarchical consensus machine* is a proposal in this direction that relies on an innovative type of stochastic reasoning and on a handover protocol. Taking advantage of incentive structures to motivate malign agents, who may deliberately seek to undermine this protocol, to behave appropriately is a peculiar aspect of consensus mechanisms in the area of blockchains. Our handover protocol is an important motivating factor for nudging malign agents in the right direction. It is conceived so that malign agents can delay a decision by our hier-

archical consensus machine but they cannot prevent it from eventually coming to one.

It is also important in this regard that the protocol guarantees only a single decision, because without this the malign agents might manipulate it to induce a fork. A blockchain fork occurs when two contradicting histories—i.e. final blocks on the same height—are simultaneously accepted.

Conceptually this seems relatively clear. The problems come from getting it to work securely in the decentralised world of agents, some of whom are malign. We identify the following issues:

A. How to create a safe but not necessarily live consensus machine? This means identifying a set of *pickets* (i.e. block-producing agents) and coming up with a model of when there is sufficient evidence among these for both them to prove that a consensus has been achieved—and similarly agree that no consensus exists without such evidence.
B. Understanding the requirements for a consensus machine to be safe. This involves understanding how malign agents can overtly misbehave to try and undermine the consensus protocol.
C. Understanding the requirements for a consensus machine to be live. This involves understanding how malign agents can misbehave via non-participation.
D. How to create a safe and live consensus machine? We would expect the liveness to come from involving many more agents in the process—in comparison to obtaining safety alone—so that it is effectively impossible for there to be enough malign ones to block the machine's progress. We will find that the combinatorics of building a safe and live system are a natural—and naturally more demanding—extension of those for building a safe one.

This paper is organised as follows. In the next section, we introduce the necessary background to make the paper self-contained. We then introduce a didactic notion of a unitary consensus machine in Sect. 3, followed by the description of a stochastic model to reason about consensus decisions in Sect. 4. We present and formalise a notion of hierarchical consensus machines in Sect. 5, discuss related work in Sect. 6, and present our concluding remarks in Sect. 7.

2 Background

2.1 Blockchains

Blockchains were initially proposed as a decentralised way to implement digital currencies and prevent double spending, i.e. the possibility that the owner of some digital currency could spend it more than once [22]. However, they have evolved into generic decentralised auditing systems that do much more than just prevent double spending. For instance, with the advent of smart contracts—programs that are executed in the context of a blockchain—a developer can define by means of a program how transactions addressed to that smart contract are to be processed [4].

A blockchain is a *decentralised stateful transaction processing system*, sometimes referred to also as a *distributed ledger*. It receives transactions from its stakeholders, decides on which of those are valid, and performs alterations to its state that record the effects of these transactions. As a decentralised system, multiple agents collaborate to implement this behaviour. In this context, the term blockchain refers not just to the state comprising the transactions and blocks, but includes the entire system including the agents operating on the state.

A blockchain orders and stores valid transactions into *blocks* which are themselves ordered, giving rise, ultimately, to a *chain of blocks* representing the history of the blockchain. In practice, however, during its operation, a blockchain—or rather its agents—manipulate a *block tree*.

A block tree is a directed, finite and acyclic rooted tree defined by a pair (V, E) where V is a set of blocks and E is a set of backward links—the root (genesis block) is the only block without an outgoing link. The backward links are implemented by embedding the cryptographic hash of its predecessor block in the header of each block. A backward link exists from block B_2 to block B_1 *iff* $hash\_pointer(B_2) = hash(B_1)$, where $hash\_pointer()$ extracts the embedded backward link from a block, and $hash()$ is the cryptographic hash function used by the blockchain. This results in a unique path from every block back to the root (B_0), because (by the properties of the hash function) it is infeasible to construct a false predecessor block with the same hash.

However, because it is possible to construct many different valid successor blocks to any existing block, it is necessary for the blockchain's agents to have a mechanism to determine unambiguously which is the "true" successor to any given block. The motivation for this paper is to provide some machinery to assist the creation of accurate consensus mechanisms.

2.2 CSP and Its Semantics

Note: In this paper we use the machine-readable ascii version of CSP syntax (CSP_M) throughout, as opposed to the typeset blackboard syntax and symbols commonly used in books and papers.

CSP is based on instantaneous actions handshaken between a process and its environment, whether that environment consists of processes it is interacting with or, some notional external observer. It enables the modelling and analysis of patterns of interaction. The books [14,27,28] all provide thorough introductions to CSP. The main constructs that we will be using in this paper are set out below.

– The processes `STOP`, `SKIP` and `DIV` respectively do nothing, terminate immediately with the signal ✓ and diverge by repeating the internal action τ. `RUN(A)` and `CHAOS(A)` can each perform any sequence of events from A, but while `RUN(A)` always offers the environment every member of `A`, `CHAOS(A)` can nondeterministically choose to offer just those members of `A` it selects, including none at all.

- `a -> P` *prefixes* `P` with the single communication `a` which belongs to the set Σ of normal visible communications. Similarly `[] x : A @ x -> P(x)` (replicated external choice) offers a choice over `A` and then behaves accordingly.
- CSP has several *choice* operators. `P [] Q` and `P |~| Q` respectively offer the environment the first visible events of `P` and `Q`, and make an internal decision via τ actions whether to behave like `P` or `Q`.
  The asymmetric choice operator `P [> Q` offers the initial visible choices of `P` until it performs a τ action and opts to behave like `Q`. In the cases of `P [] Q` and `P [> Q`, the subsequent behaviour depends on what initial action occurs.
- `P \ X` (hiding) behaves like `P` except that all actions in `X` become (internal and invisible) τs.
- `P [[ R ]]` (renaming) behaves like `P` except that whenever `P` performs an action `a`, the *renamed* process must perform some `b` that is related to `a` under the relation `R`. `R` is specified using the CSP_M mapping syntax.
- `P [| A |] Q` is a *parallel* operator under which `P` and `Q` act independently except that they have to agree (i.e. synchronise or handshake) on all communications in `A`. A number of other parallel operators can be defined in terms of this, including `P ||| Q = P [||] Q` in which no synchronisation happens at all.

There are also other operators such as `P ; Q` (sequential composition), `P /\ Q` (interrupt) and `P [| A |> Q` (throwing an exception) for passing control from one process `P` to a second one. `P /\ Q` hands over control when `Q` performs a visible action, so that the handover if instigated by Q. In `P [| A |> Q` it is instigated by `P` performing an exception event `a` from the set `A`.

It is always asserted that the meaning, or semantics, of a CSP process is the pattern of externally visible communication it exhibits. As shown in [27,28], CSP has several styles of semantics, that can be shown to be appropriately consistent with one another. In this paper, we are concerned with *behavioural* semantics: CSP processes are identified with sets of observations that might be made from the outside. The best known behavioural models of CSP are based on the following types of observation: *Traces* are sequences of visible communications a process can perform. *Failures* are combinations (s, X) of a finite trace s and a set of actions that the process can refuse in a *stable* state reachable on s. A state is stable if it cannot perform τ. *Divergences* are traces after which the process can perform an infinite uninterrupted sequence of τ actions, in other words diverge. The models are then:

- `T` in which a process is identified with its set of finite traces;
- `F` in which it is modelled by its (stable) failures and finite traces;
- `FD` in which it is modelled by its sets of failures and divergences, both extended by all extensions of divergences: it is *divergence strict*.

2.3 FDR

FDR [10,26–28] is a refinement checker between finite-state processes defined in CSP. First created in the early 1990's it has been regularly updated since. The latest version is FDR4.[1]

It uses CSP_M, the machine-readable version of CSP, which has been extended with a functional programming language related to Haskell. This enables the user to define complex networks and data operations succinctly, and to create functions that, given abstract representations of structures or systems, can automatically generate CSP networks to implement and check them. Perhaps the best-known example of this is the Security Protocol checker Casper [21] which, given an abstract representation of a cryptographic protocol and some security objectives for it, generates a CSP script which checks to see if the objectives are met. In a similar vein, compilers have been written from other notations to CSP such as Statecharts [13] and shared-variable programs (see Chapters 18 and 19 of [28]). A survey of the most important practical applications of FDR can be found in [2].

FDR is most often used to check refinements of the form `Spec [X= Impl`, where `Spec` is a process representing a specification in one of the standard CSP models `X`, usually traces, stable failures or failures-divergences. `Impl` is a CSP representation of the system being checked. To check whether a process `Impl` satisfies a particular property, `Spec` is constructed to represent the most general process (in the relevant model) exhibiting the required property.

FDR supports a number of techniques for attacking the state explosion problem, including hierarchical compression and symmetry reduction [11]. The algorithms underpinning FDR are set out in [10,27–29].

3 The Unitary Consensus Machine

One of the main problems in designing a blockchain is devising how to select a unique successor for a given block; the initial (often termed *genesis*) block is preagreed between agents and assumed to exist, however there may be more than one plausible candidate for any subsequent block. This problem is often solved by a protocol that determines whether a block is *final* in blockchain terminology. Typically, the *finality* of a block is determined by a universally known, though potentially randomly selected, committee of agents, which we call *pickets*, that engage in a protocol by which they reach a *consensus* on the successor of a given block. We call such a system composed of interacting pickets that solves the problem of determining the finality of a block a *consensus machine*. Since blockchains are systems intended to cope with adversarial behaviour (coming from untrusted parties), these machines are designed to tolerate a certain proportion of malign agents. That is, the expected overall behaviour emerges from the interaction of pickets in spite of possible misbehaviour by malign agents

[1] Available at https://cocotec.io/fdr/.

amongst them. The notions described here can also be applied to the problem of reaching consensus for more general distributed systems.

We first illustrate how a *unitary consensus machine* works, i.e., how a single set of pickets can interact to reach consensus. Later, we build on this illustration to propose our *hierarchical consensus machine*. The informal description that we provide here illustrates the mechanism used by the hierarchical protocol we propose later.

Let P be a set of pickets, D be a set of possible decision values that the pickets are trying to reach consensus on, and $M \subseteq \mathcal{P}(P)$ the *decision sets* such that agreement by any set $m \in M$ commits the system to the agreed decision, where M is superset closed and contains P and $\mathcal{P}(S)$ gives the power set of S. Broadly speaking, the unitary consensus machine works as follows. For a given run of this machine P, M, and D are fixed and well-known. Each picket $p \in P$ locally decides on a single value $v_p \in D$ and broadcasts this chosen value. We assume that pickets have well-known public keys as part of agreed cryptographic signature schemes so they can create unforgeable digitally signed messages. The set $m_{o,v}$ denotes the set of pickets that have chosen value v according to the messages received by observer o. If $m_{o,v} \in M$, observer o knows that the machine has decided on value v. In this paper, we focus on a restricted scenario involving a single run of the machine, i.e., having pickets decide on a value a single time. There is no issue in extending this for a series of decisions where each is properly made before the next one starts.

We note that since the evidence for a decision will be an agreed and signed decision by sufficient agents for some $m \in M$, no-one can dispute a properly formed one. We require that whatever decision is made is agreed with by at least one benign agent that follows all the rules: this will be a property of M.

We require well-behaved consensus machines to additionally respect two properties:

- **Safety:** For observers o_1, o_2 and $v_1, v_2 \in D$, if $m_{o_1,v_1} \in M$ and $m_{o_2,v_2} \in M$, it must be the case that $v_1 = v_2$.
- **Liveness:** After observing the consensus machine run for stabilisation time t, an observer o is able to construct a set $m_{o,v}$ such that $m_{o,v} \in M$.

Intuitively speaking, the safety property forbids the machine from deciding on two distinct values (on the same run), whereas the liveness property ensures that the machine eventually decides on a value. Note that the liveness property implicitly accounts for enough pickets agreeing on a given value but also for their decision being conveyed in a timely manner.

We assume that malign agents can deviate from the expected picket behaviour arbitrarily. For instance, they could send as their chosen value v_1 to observer o_1 while sending a distinct value v_2 to observer o_2—this double choice is a common Byzantine behaviour expected of such malign agents.

The safety property depends on M considering the benign and malign agents in P. For instance, if sets $m_1, m_2 \in M$ are crafted so that there is no benign picket that is part of both m_1 and m_2, these two committees could decide on two distinct values on the same run. Thus, (i) any two sets in $m_1, m_2 \in M$ must have

an overlapping benign picket to achieve the safety property. We can show (i) by contradiction. Let us assume that sets $m_{o_1,v_1} \in M$ and $m_{o_2,v_2} \in M$ with $v_1 \neq v_2$ were constructed. Then, by (i), $p' \in m_{o_1,v_1}$ and $p' \in m_{o_2,v_2}$, which implies that the benign picket p' choose two values v_1 and v_2, a contradiction.

To ensure liveness, one must assume or enforce that: (a) there is some *stabilisation time* by which point messages from benign pickets are delivered; and (b) a set $m \in M$ of benign pickets chooses the same value (in time for stabilisation); the stabilisation time is required to move away from impossibility results [9]. While (b) ensures a decision is made, (a) ensures that an observer can witness this decision. For our minimal unitary consensus machine presented in this section, we assume that such a set m exists as pickets are making their choice. In practice, however, if no set $m \in M$ could be constructed—when, for instance, pickets choose different values—the protocol would have a recovery mechanism by which pickets would choose another value to try and build such a m; the protocol would be constructed so that pickets converge into an agreed value after some time.

It is crucial to understand the dichotomy between safety and liveness in the setting we study: one can be more tolerant of malign pickets' involvement when crafting an M that is safe but not live as opposed to one that is safe and live; this observation follows from properties (i) and (b). There are decision sets M that abide by property (i) and yet cannot satisfy (b). For instance, we could have an M that abide by (i) but all its member include a malign picket. In these cases, the participation of benign nodes in the members of such an M ensure decisions are safe. However, the presence of malign pickets may cause a decision to never be reached as they can refuse to participate in the consensus protocol. This observation is one of the main principles guiding the design of our *hierarchical consensus machine*.

In the context of blockchains, a consensus machine is meant to determine the *true/canonical chain* by repeatedly picking successor blocks—and pruning the block tree in the process. These blocks, and the transactions that they contain, represent transitions in the state of the blockchain. They can account, for instance, for a transfer of digital currency or the execution of some code (i.e. via a smart contract). Thus, assuming that these transactions are deterministic, the consensus machine also determines the canonical sequence of states of the blockchain.

Blockchains are frequently set up with incentive and penalty structures that are designed to persuade the malign agents to follow the rules. We categorise malign behaviour as follows:

1. Overt malign behaviour. Making contributions to the central discussions and protocols of a chain or other decentralised system that will be seen and recognised as malign. Unless this wins votes or similar, it will quickly be recognised and the perpetrator punished.
2. Covert malign behaviour. Producing non-compliant structures that are kept hidden and only perhaps revealed later. For example developing a fork alongside the true chain.

3. Non-participation. Failing to make contributions that are expected of a good agent and thereby denying some correct action the majority it needs. The main issues with this is that it is harder to penalise because a good agent may encounter communication failures, a phenomenon that can also mean confusion about how an apparently non-participating agent should be interpreted. It is fairly standard to make gossiping assumptions about communications in blockchains to resolve such confusion.

The sorts of incentive structures implemented by blockchains are another important factor that guided the design of our hierarchical consensus machine. In particular, non-participation failures may cause the need to transfer control from one unitary consensus machine to another, in order to achieve overall liveness.

4 Stochastic Decisions

The security analysis of blockchains is usually predicated upon some assumed distribution of malign agents. So, we use probability to assemble sets of pickets and produce decision sets M. In this section, we discuss a central case of how this can support the picketing model. We assume that pickets are drawn from an agent population U where the probability that a randomly chosen agent is benign is p, and that they are selected independently and randomly from U so that the number of benign and malign pickets that make any decision set is governed by a binomial distribution, that is, $\binom{n}{k}p^k(1-p)^{n-k}$ gives the probability of having k benign agents when selecting n agents from U. Given this assumption, it is relatively easy to compute how likely it is that at most r out of n picket selections are benign: $F(p,n,r) = \sum_{i=0}^{r}\binom{n}{i}p^i(1-p)^{n-i}$.

Based on these s, we propose the idea of *stochastic impossibility*: an event so unlikely that in the whole history of a system it is very unlikely that one will happen, to the extent that it can be disregarded. This concept is parameterised by a *insignificance threshold* ϵ and an event that happens with probability $\xi \leq \epsilon$ is termed *stochastically impossible*. One might regard a one-in-a-million chance as small enough, but if many (say a million) choices are going to be made a year (approximately one every 30 s) it is clearly is not enough if a single one can corrupt a system. We believe that the $\epsilon = 10^{-18}$ is a reasonable starting point; in terms of the normal distribution, this value is close to $9\sigma\,(\approx 10^{-19})$, where σ is the standard deviation, namely, the cumulative probability from $\mu + 9\sigma$ to infinity, where μ is the mean. This sort of σ-multiplier analysis is used in finance to model risk [7], and is justified as a consequence of the probabilistic laws of large numbers.

We can now understand how to create the decision thresholds M described earlier. Until now, we have informally referred to the groups of pickets selected to make our decisions as *sets*. However, because a given agent can validly be selected more than once (randomly *with replacement*) when assembling decision "sets", these groups are actually *multisets* (bags). This also explains why the binomial distribution is the appropriate model to use when computing the probability that at least a certain specified number of pickets in such a group are benign. In

a population U of agents each with independent probability p of being benign, a randomly drawn sub-multiset of pickets $P \subseteq U$ is said to have (stochastically certainly) at least $k+1$ benign agents if $F(p,k,|P|) < \epsilon$; this inequality means that having at most k benign agents is stochastically impossible. For fixed p, k, and ϵ, we can calculate the smallest $|P|$ so that at least $k+1$ agents are benign; let us call this threshold value $td(p,k,\epsilon)$. Given that a multiset of pickets P where $|P| = td(p,k,\epsilon)$ has at least $k+1$ benign agents, any sub-multiset $m \subseteq P$ such that (1) $|m| \geq |P| - (k+1) + b$ includes at least b benign agents.

To achieve safety via (i), we need to have more than half of the $k+1$ benign agents in any $m \in M$. So, by using $b = k/2 + 1$ in (1), we have that $|m| \geq |P| - k + \lceil k/2 \rceil$, where $k/2$ is integer division. Therefore, for $M = \{m \subseteq P \mid |m| \geq |P| - k + \lceil k/2 \rceil\}$, we have that property (i), and safety, is satisfied, modulo stochastic certainty.

To achieve liveness via (b), we need to have (2) $|m| \leq k+1$, namely, at least a decision set that requires (modulo stochastic certainty) only the participation of benign agents for agreement. Thus, to have safety and liveness, one has to satisfy (1) and (2). The inequality (I) $|P| \leq \lfloor 3k/2 \rfloor + 1$ has to be satisfied in order to ensure both (1) and (2). This inequality gives the bounds that are usually referred to in consensus literature [8].

Table 1 illustrates some examples of calculation for the largest k such that $F(p,k,n = |P|) < \epsilon$, for some values of n (number of selected agents) and p (benignity probability) and where is fixed $\epsilon = 10^{-18}$. This calculation is analogous to the one presented. Red entries in the top left are where even seeing all agents agreeing does not prove this, as it is deemed possible that all the agents are malign, namely, for these values of p, n, and ϵ, there is no k such that $F(p,k,n) < \epsilon$. In purple areas, we have that k and n satisfy (I). We can achieve safety for all but the red cells in the upper left corner. However, safety and liveness can only be achieved for the purple cells in the right bottom corner. This pattern illustrates that achieving both safety and liveness requires larger sets of pickets and decision sets in comparison to achieving safety alone. For example, with $p = 0.95$ and $n = 50$, we have that $k = 25$. So, we have at least 26 benign agents amongst the 50 randomly and independently selected. Thus, to ensure safety, we can choose decision sets $m \subseteq P$ such that $|m| \geq 38$. Since (I) does not hold for $n = 50$ and $k = 25$, we cannot obtain safety and liveness. On the other hand, for $p = 0.95$ and $n = 100$, we have that $k = 66$, in which case (I) holds. For this case, we can have decision sets $m \subseteq P$ such that $|m| \geq 67$ to achieve liveness and safety. Smaller pickets and decision sets should also allow for more efficient agreement given that fewer agents need to actively take part in the protocol; this principle was one of the main drives in designing our hierarchical consensus machine.

Usually our systems do rely on decisions being made, and usually the systems are more efficient if they can persuade malign participants to contribute mostly as though they were good. Indeed for a consensus machine with a smaller set of pickets to deliver results, this is necessary. To achieve this they need three things: firstly strong incentives on agents not to misbehave and to participate

Table 1. Examples of k for different combinations of p and n, and fixed $\epsilon = 10^{-18}$. The p values were chosen with the consensus bounds of $> 2/3$ in mind.

0.66	⋆	⋆	0	3	6	14	22	70	177	290	525
0.75	⋆	0	3	7	12	22	32	91	218	351	624
0.8	⋆	1	5	10	16	27	39	104	243	387	*682*
0.85	⋆	3	8	14	20	33	46	118	*269*	*425*	*742*
0.9	0	6	12	19	26	40	55	*134*	*298*	*466*	*807*
0.95	3	10	17	25	33	50	*66*	*153*	*331*	*512*	*878*
p/n	20	30	40	50	60	80	100	200	400	600	1000

constructively, secondly a decision making mechanism that prevents the malign from inducing a bad decision, and thirdly a fallback mechanism that can force correct decisions (i.e. is both safe and live) when needed, all be it at the cost of lower efficiency. The last of these should convince opponents that they will not be able to permanently disrupt the system. The worst they can achieve is complication and delay. One cannot reasonably prevent the malign from covert mischief, but overtly saying the wrong thing or not doing what they are meant to will attract penalties and bans. The main motivation for the hierarchical consensus machine idea introduced next is providing the required fallback mechanism. It allows us to initiate a decision on the assumption that (most of) the malign agents participate normally in the knowledge that the carefully-picked (safe) decision sets will prevent a bad decision from being made; the (live) fallback allows a decision to be forced even when malign agents do not participate.

5 Formalising Hierarchical Consensus Machines in CSP

We have already described how a consensus machine proceeds when it consists of a single set of pickets synchronising in a rather abstract sense. We have also described how to pick decision sets so that one can achieve safety and liveness using a type of stochastic reasoning. In this section, we present a *hierarchical consensus machine*, let us call it *HM*, that is in itself a combination of two (sub-)consensus machines, let us call them, G and H. The machine G is safe and efficient, whereas H is less efficient but it is both safe and live; as explained in the previous section, the difference in efficiency comes from the size of picket and decision sets that are necessary to achieve these properties. In achieving safety without liveness, G can enter a situation very similar to the well-known phenomenon of deadlock, when malign agents refuse to take part and agree on a value. Deadlock is not normally an acceptable behaviour of a complete system, and certainly not in a blockchain. We propose a way to recover from such a deadlock in G by letting H take over. Specifically, we show how control of a decision-making procedure can be handed from one machine to the other. Despite G not being live, *HM* still is so thanks to H and the handover protocol we propose.

When passing decision making from G to H, the transition might come because the agents in G have the evidence that G will not be able to decide, or because malign agents in G fail to participate—in the latter case, G will not reach a decision but its agents are unable to determine that it will not. In both cases, we need to be careful that control will not be passed to H when some agents in G are already committed to a value, or at least that, in this case, H decides on the same committed value. So, our protocol does not prevent H and G both issuing decisions, but ensures that if they do, they are the same.

The more difficult of these cases is where the pickets in H take over on their own initiative. That is because if G's agents themselves decide to hand over, it will be because there is agreement to do so. Handing over to H means that G has not made the decision, and none of G's agents can validly believe it has, as that would be inconsistent with the agreement to hand over. When taking over from G, the H process does not have an immediate global effect on all the agents of G, so a decision may still be made later by G.

Our formulation is inspired by the large body of work on process algebra: understanding bodies of agents that run concurrently and interact by forms of synchronisation. There is an interesting analogy here with process algebra. CSP, particularly in later versions [27,28], has a number of ways in which one process can pass control to another. The throw operator `P [| A |> Q` runs like P until it throws an exception in the set A, which causes it to run like Q. On the other hand the interrupt operator `P /\ Q` has P run, but if Q performs any visible action it takes over.

We present and formalise in CSP two models for *HM*. The *abstract model* represents the behaviour of each machine G and H as a single CSP process. It abstractly depicts what is the expected emergent behaviour from their respective implementation each of which as an interactive distributed set of pickets. The main step of this abstraction is that the component consensus machines G and H are deemed to take an action only once there is agreement (in the sense we have already discussed) on the action. The *distributed model*, on the other hand, demonstrates precisely how the emergent behaviour of each machine can be realised in terms of such a set of pickets.

In other words the abstract model describes how we expect the protocol to work in every implementation, but the way in which the sequential processes it contains are implemented by decentralised collections implemented by G and H are not laid down. The distributed model illustrates one way of realising this.

The protocol we present here has much in common with mutual exclusion. We want to prevent something akin to a race condition. An obvious question is whether we could use a simple mutex between G and H and only allow one to make the decision. The answer is no: it is part of the make-up of G that it can deadlock at any time. If it were to seek the right to make the decision—via the shared mutex—but then deadlock, then *HM* would deadlock too; contrary to our specification.

5.1 Abstract Model

In the abstract model, each of G and H is modelled as a single CSP process, and they communicate via shared storage locations each of which is also represented as a CSP process and each machine has two locations it can write to. Intuitively speaking, machine G comes to a decision in a two-step process. It first *commits* to (i.e. pre-decides on) a value by writing it on its first location and then it *decides* on this value by writing to its second location. Before these writes it checks whether H has started by looking for a *started* signal written to H's first location. If at any point it detects that H has started, it stops by choice. After a timeout has elapsed, H starts. It initially checks whether G has come to a decision already. If so, it reaffirms that decision. Otherwise, it signals it has started its decision making process by writing a *started* signal value on its first storage location. If no value has been committed to by G at that point, H proceeds to make its own decision. Otherwise, again, it just echoes G's decision.

Machines G and H rely on storage locations to communicate and convey (pre-)decisions. The datatype `values` denotes the possible values stored in these locations: `D1` and `D2` are (pre-)decisions whereas `quiet`, `start` and `null` denote machine statuses. Locations are identified by elements in `location`. Locations 1 and 3 are controlled (i.e. written) by machine G whereas 2 and 4 are controlled by machine H. Channels `read`, `write1` and `write2` are used to manage locations whereas `stepG`, `stepH`, and `timeoutstep` denote internal actions of these machines. Finally, channel `decision` is used to communicate (pre-)decisions made by them.

```
datatype values = quiet | started | D1 | D2 | null

locations = {1..4}

channel read, write1: locations.values
channel stepG, stepH, timeoutstep

channel decision:{1,2}.{D1,D2}
```

The storage locations are defined by the following two processes. Writing to and reading from these locations are not atomic events. When a value y is written to location i (via `write1.i?y`) the storage goes into a non-deterministic state in which it allows for a read to retrieve the old value x. The event `write2.i` signals to this location that the value y has been properly written at which point reads deterministically return y. This non-determinism captures (i.e. abstracts) the asynchrony of the distributed system: the write begins when the decision is known somewhere and it ends when it is known at most of the network.

```
Store(i,x) = read.i!x -> Store(i,x)
    [] write1.i?y -> StoreND(i,x,y)

StoreND(i,x,y) = (read.i.x -> StoreND(i,x,y)
```

```
                |~| read.i.y -> StoreND(i,x,y))
    [] write2.i -> Store(i,y)
```

We abstract away all activities of G and H except the steps they need to make to record the decision they make and the steps they need to record and coordinate it. For modelling purposes we assume here that G makes decision `D1` and H makes `D2` unless it is forced to follow G's decision because it cannot be sure G will not make a decision.

The machine G's behaviour is defined by the following CSP processes, with initial state given by `G`. As `G0`, it reads the status of machine H via location 2. If H has started already, it stops. Otherwise, if H is quiet, as process `G1`, it signals a pre-decision on value `D1` by writing it to Location 1. If H is still quiet at that point, it consolidates this pre-decision with event `write2.1`. As process `G2`, it reads the status of H for the last time, before issuing a final decision on `D1` as process `G3`. Note that the parallel combination of `G0` and `CHAOS` in `G` captures G's incompleteness by allowing it to deadlock at any point.

```
G0 = (read.2.quiet -> stepG -> G1
 [] read.2.started -> STOP)

G1 = write1.1.D1 ->
     (read.2.quiet -> write2.1 -> stepG -> G2
     [] read.2.started -> STOP)

G2 = read.2.started -> STOP
     [] read.2.quiet -> stepG -> G3

G3 = decision.1.D1 -> write1.3.D1 -> write2.3 -> STOP

G = G0 [|Events|] CHAOS(Events)
```

The machine H's behaviour is defined by the following CSP processes, with initial state given by `H`. As `H`, it reads whether machine G has come to a decision by reading Location 3. If it detect a decision, it re-asserts this decision by writing `D1` to Location 4. Otherwise, it interprets that a timeout has occurred and it moves on to make its own decision. As `H1`, it signals that it has started its decision making process by writing `started` to Location 2. As `H2`, it checks whether machine G has started at all. If it has, H re-asserts the pre-decision made by G—i.e., by writing `D1` to Location 4. Otherwise, it proceed by making its own decision by writing `D2` instead. Both of these decisions are captured by process `H3`.

```
H = read.3.null -> timeoutstep -> H1
    [] read.3.D1 -> write1.4.D1 -> write2.4 -> STOP

H1 = write1.2.started -> write2.2 -> stepH -> H2
```

```
H2 = read.1.null -> stepH -> H3(D2)
     [] read.1.D1 -> H3(D1)

H3(d) = decision.2.d -> write1.4.d -> write2.4 -> STOP
```

The hierarchical consensus machine behaviour is given by `System`. Note how machines H and G are interleaved in `Machine` and they rely on storage locations in `Locations` to interact as we discussed.

```
Locations = Store(1,null) ||| Store(2,quiet)
                ||| Store(3,null) ||| Store(4,null)

Machines = G ||| H

System = Machines [|{|read,write1,write2|}|] Locations
```

We expect this abstract hierarchical consensus machine to be *safe and live*. By safe, we mean that if it comes to a decision, it decides on a single value, that is, each machine might even come to their own decision but their value must match. By live, we mean that `System` must not deadlock before a decision is made. We capture these two requirements by a refinement expression in CSP's stable failures model as follows.

```
Decisions = {write1.3.d, write1.4.d | d <- {D1,D2}}
Decision1 = {write1.3.d, write1.4.d | d <- {D1}}
Decision2 = {write1.3.d, write1.4.d | d <- {D2}}

DSystem = System \ diff(Events,Decisions)

Spec =(|~| x:Decision1 @ x -> CHAOS(Decision1))
        |~|
       (|~| x:Decision2 @ x -> CHAOS(Decision2))

assert Spec [F= DSystem
```

The refinement expression is built around decision events: all the decision events are members of `Decisions`, the decision events for value `D1` are members of `Decision1`, and the events for value `D2` are in `Decision2`. The specification process `Spec` allows a decision to be made on `D1` and `D2` initially. Once such a decision is made, only events deciding on that value are allowed be performed. Note that this process is not allowed to deadlock initially. Thus, the proposed refinement expression ensures that the behaviour of the system when projected onto decision events—given by `DSystem`—offers some decision event initially and stick to that decision value subsequently. We have used FDR to validate this refinement expression.

5.2 Distributed Model

The abstract model is useful from an analysis perspective: one can analyse the handover protocol itself while not needing to examine the implementation of each machine as a collection of interactive agents and the issues arising from such an implementation. Instead, issues with just the handover protocol itself can be identified and fixed. We can then argue either that a given approach to building the individual machines G and H will meet this model by construction, or test it by building a more detailed, distributed model in CSP for FDR.

In our model, each machine is a distributed system implementing a protocol that attempts to reach consensus in the presence of Byzantine agents. Intuitively speaking, our hierarchical machine works as follows. Machine G starts and tries to come to a decision on a unified value. After some appropriate amount of time—enough to allow G to come to a decision if agents can agree on a value—machine H starts. It checks whether machine G has *committed* to a value, i.e., it has pre-decided on it but might not have gathered enough evidence to properly decide on it. If so, machine H decides on that value. Otherwise, the agents in H are free to choose a value of their own. Like the abstract model, these machines communicate local decisions via storage locations.

Our more detailed CSP model is parameterised by some global functions. `VALUES` gives the universe of decision values, and `NODES` are the agent identifiers. For machine `m`, `N(m)` gives its number of agents, `MNODES(m)` are its agent identifiers, `THRESHOLD(m)` gives the level of agreement (i.e. how many agents) that is required for reaching consensus, `G(m)` gives the number of good agents, with `GOOD(m)` and `BAD(m)` identifying the good and malign agents in the machines, respectively. In the following, we describe in detail our CSP model.

```
datatype MACHINES = g | h

channel value : MACHINES.NODES.VALUES
channel prewrite, write : MACHINES.NODES.MACHINES.NODES.VALUES
channel setup_prewrite, setup_write : MACHINES.NODES.VALUES
channel decision : MACHINES.VALUES
channel decide : MACHINES.NODES.VALUES
channel timeout : MACHINES.NODES.MACHINES.NODES
channel end_round
```

We use event `value.m.n.v` to represent that the agent `n` in machine `m` has chosen as its decision value `v`, event `setup_prewrite.m.n.v` (`setup_write.m.n.v`) to signal that agent `n` in machine `m` has pre-decided (decided) on value `v`, and event `pre-write.m.n.mm.nn.v` (`write.m.n.mm.nn.v`) as a way to communicate to agent `nn` in machine `mm` that agent `n` in machine `m` has pre-decided (decided) on value `v`. The event `decision.m.v` is used to signal that machine `m` has decided on value `v`, whereas `decide.m.n.v` are convey that agent `n` in machine `m` has (locally) decided on value `v`. The event `timeout.m.n.mm.nn` denotes that agent `nn` in machine `mm` timed out when trying to read the decision from agent `n` in machine `m`. The event `end_round` is a modelling device used to signal that

machine G has had enough time to come to a decision and that machine H is now taking over.

```
EmptyPreWriteLocation(n,m) =
    setup_prewrite.m.n?v -> FullPreWriteLocation(n,m,v)

FullPreWriteLocation(n,m,v) =
    prewrite.m.n?mm?a:MNODES(mm)!v -> FullPreWriteLocation(n,m,v)
```

The process `EmptyPreWriteLocation(n,m)` is a storage location that stores the pre-decision of agent `n` in machine `m`; each agent has such a location that it controls. It is a single-write multiple-reads one-place buffer.

```
EmptyWriteLocation(n,m) =
  setup_write.m.n?v -> FullWriteLocation(n,m,v)
  []
  timeout.m.n?mm?a:MNODES(mm) -> EmptyWriteLocation(n,m)

FullWriteLocation(n,m,v) =
  write.m.n?mm?a:MNODES(mm)!v -> FullWriteLocation(n,m,v)
  []
  decide.m.n.v -> FullWriteLocation(n,m,v)
```

The process `EmptyWriteLocation` is also a storage location that behaves similarly to the previous one. It stores decisions instead of pre-decisions. Moreover, it offers a `timeout` event if the location is empty—it allows agents reading from it to experience a timeout—and it uses the `decide` event to communicate the local decision of this agent.

```
GNode(n) =
  value.g.n?v -> setup_prewrite.g.n.v ->
    if v == 0 then PreWrite(n,g,{n},1,0,0)
    else if v == 1 then PreWrite(n,g,{n},0,1,0)
    else PreWrite(n,g,{n},0,0,1)
```

The control behaviour of agent `n` in machine G is given by process `GNode(n)`. We design the agents so that they choose their local decision value independently (captured by event `value`) but they will come together, or not, to certify a unified decision. Once a value is chosen, it is written to the agent's pre-decision storage (via event `setup_prewrite`).

```
PreWrite(n,m,vs,c0,c1,c2) =
    (prewrite.m?a:diff(MNODES(m),vs)!m.n?v ->
        if v == 0 then PreWrite(n,m,union({a},vs),c0+1,c1,c2)
        else if v == 1 then PreWrite(n,m,union({a},vs),c0,c1+1,c2)
        else PreWrite(n,m,union({a},vs),c0,c1,c2+1))
    []
```

```
    (timeout.m?a:diff(BAD(m),vs)!m.n ->
        PreWrite(n,m,union(vs,{a}),c0,c1,c2))
    []
    (vs == MNODES(m) &
      if c0 >= THRESHOLD(m) then setup_write.m.n.0 -> EndOfRound
      else if c1 >= THRESHOLD(m) then setup_write.m.n.1 ->
        EndOfRound
      else if c2 >= THRESHOLD(m) then setup_write.m.n.2 ->
        EndOfRound
      else EndOfRound)

      EndOfRound = end_round -> SKIP
```

The `PreWrite` process describes how an agent reads the pre-decisions of other agents in order to come to its own local decision. Once the agent has received a pre-decision or a timeout from all nodes, it goes on to either locally decide on a value or to conclude the decision making process without deciding on a value. If it has seen enough pre-decisions supporting value v—for instance, for $v == 0$, this is captured by condition `c0 >= THRESHOLD(m)`—the agent locally decides on v, writing this value to its decision storage location (via event `setup_write`). Note how the agent only accepts timeouts from malign agents; we assume that good agents deliver messages reliably and in a timely way. The process `EndOfRound` signals that machine's G time to come to a decision has elapsed, at which point, the agent terminates.

```
GGoodNode(n) = (GNode(n) [|{|setup_prewrite, setup_write|}|]
    (EmptyWriteLocation(n,g) ||| EmptyPreWriteLocation(n,g)))
```

A benign agent in machine G is a parallel process—given by process `GGoodNode`—that combines its storage locations and its control behaviour.

```
GoodAlpha(n,m) =
    Union({{| value.m.n, setup_prewrite.m.n, setup_write.m.n,
            decide.m.n, prewrite.m.n.mm.a, prewrite.mm.a.m.n,
            timeout.mm.a.m.n, timeout.m.n.mm.a, write.m.n.mm.a,
            write.mm.a.m.n, end_round | mm <- MACHINES,
            a <- MNODES(mm), (a != n or mm != m) |}})
```

`GoodAlpha(n,m)` gives the alphabet of the benign agent `n` in machine `m`.

```
HNode(n) =
    end_round ->
        Reader(n,{},0,0,0)

Reader(n,vs,c0,c1,c2) =
    (write.g?a:diff(MNODES(g),vs)!h.n?vv ->
        setup_prewrite.h.n.vv ->
```

```
        if vv == 0 then setup_write.h.n.0 -> EndOfRound
        else if vv == 1 then setup_write.h.n.1 -> EndOfRound
        else setup_write.h.n.2 -> EndOfRound)
    []
    (timeout.g?a:diff(MNODES(g),vs)!h!n ->
        Reader(n,union(vs,{a}),c0,c1,c2))
    []
    (vs == MNODES(g) &
      value.h.n?vv -> setup_prewrite.h.n.vv ->
        if vv == 0 then PreWrite(n,h,{n},1,0,0)
        else if vv == 1 then PreWrite(n,h,{n},0,1,0)
        else PreWrite(n,h,{n},0,0,1))
```

The control behaviour of a benign agent in machine H is given by process `HNode(n)`. The initial `end_round` event and the requirements that we impose on the way in which agents synchronise on this event means that the agents of machine H only start after the agents of machine G have finished with their decision making interactions. This behaviour captures the assumption that agents have a reasonably synchronised clock and that they can come to a decision within a bounded time frame.

Once started, the agent's control behaviour in machine H is given by `Reader`. This process reads the local decisions made by agents in G. If one of them has decided on a given value—which means that machine G has committed to that value—we require that the agent in H decide on the same value. This behaviour ensures that if both machines come to a decision, they must agree on their decided value.

If no agent of G has decided on a value, the agents in H are free to choose their local decision values, and they move on to behave like process `PreWrite` to try and come to a unified decision as already mentioned.

```
HGoodNode(n) = (HNode(n) [|{|setup_prewrite, setup_write|}|]
    (EmptyWriteLocation(n,h) ||| EmptyPreWriteLocation(n,h)))
```

Similar to benign agents in G, a benign agent in H is a parallel combination of its control behaviour and storage locations as per process `HGoodNode`.

```
BadNode(n,m,c0,c1,c2) =
  timeout.m.n?mm?a:GOOD(mm) -> BadNode(n,m,c0,c1,c2)
  []
  (STOP
  |~|
  (prewrite.m.n.m?a:diff(MNODES(m),{n})?v ->
    BadNode(n,m,c0,c1,c2)
  []
  prewrite.m?a:diff(MNODES(m),c0)!m.n.0 ->
    BadNode(n,m,union(c0,{a}),c1,c2)
  []
```

```
  prewrite.m?a:diff(MNODES(m),c1)!m.n.1 ->
    BadNode(n,m,c0,union(c1,{a}),c2)
  []
  prewrite.m?a:diff(MNODES(m),c2)!m.n.2 ->
    BadNode(n,m,c0,c1,union(c2,{a}))
  []
  card(c0) >= THRESHOLD(m) &
    (write.m.n?a.b!0 -> BadNode(n,m,c0,c1,c2)
    [] decide.m.n.0 -> BadNode(n,m,c0,c1,c2))
  []
  card(c1) >= THRESHOLD(m) &
    (write.m.n?a.b!1 -> BadNode(n,m,c0,c1,c2)
    [] decide.m.n.1 -> BadNode(n,m,c0,c1,c2))
  []
  card(c2) >= THRESHOLD(m) &
    (write.m.n?a.b!2 -> BadNode(n,m,c0,c1,c2)
    [] decide.m.n.2 -> BadNode(n,m,c0,c1,c2))))
```

The malign agent `n` in machine `m` is modelled by process `BadNode(n,m)`. These agents can exhibit Byzantine behaviour but they are not allowed to behave completely arbitrarily: there are still some actions which these adversaries cannot perpetrate against benign agents. For instance, it can only offer event `decide` if it has gathered enough support for the corresponding decision—i.e., it cannot create a spurious local decision. This abstraction accounts for the following behaviour: a local decision by an agent must be associated with enough supporting evidence—in the form of pre-decisions—which are cryptographically signed by the agents generating that evidence. We assume malign agents cannot break cryptographic primitives and, thus, they cannot forge signatures by other agents. On the other hand, malign agents can pre-decide on more than one value, or even refuse to serve a request for a (pre-)decision.

```
BadAlpha(n,m) = {| prewrite.m.n.mm.a, prewrite.mm.a.m.n,
    write.m.n.mm.a, write.mm.a.m.n, decide.m.n, timeout.m.n.mm.a
    | mm <- MACHINES, a <- MNODES(mm), (a != n or mm != m) |}
```

The alphabet of malign agent `n` in machine `m` is given by `BadAlpha(n,m)`.

```
AlphaBadNodes(m) = Union({BadAlpha(i,m) | i <- BAD(m)})
BadNodes(m) = || i : BAD(m) @
    [BadAlpha(i,m)] BadNode(i,m,{i},{i},{i})

AlphaGoodNodes(m) = Union({GoodAlpha(i,m) | i <- GOOD(m)})
GGoodNodes = || i : GOOD(g) @  [GoodAlpha(i,g)] GGoodNode(i)
GNodes = GGoodNodes [ AlphaGoodNodes(g)
    || AlphaBadNodes(g) ] BadNodes(g)
```

```
HGoodNodes = || i : GOOD(h) @  [GoodAlpha(i,h)] HGoodNode(i)
HNodes = HGoodNodes [ AlphaGoodNodes(h)
    || AlphaBadNodes(h) ] BadNodes(h)

Nodes = GNodes [union(AlphaGoodNodes(g),AlphaBadNodes(g))
            || union(AlphaGoodNodes(h),AlphaBadNodes(h))] HNodes
```

The processes `GNodes` and `HNodes` capture the behaviour of machines G and H, respectively, whereas `Nodes` captures how they interact to implement the handover protocol. In these processes, the appropriate agents run in parallel and they are required to synchronise on shared events.

```
Decider(m,c0,c1,c2) =
  decide.m?a:diff(MNODES(m),c0)!0 -> Decider(m,union({a},c0),c1,c2)
  []
  decide.m?a:diff(MNODES(m),c1)!1 -> Decider(m,c0,union({a},c1),c2)
  []
  decide.m?a:diff(MNODES(m),c2)!2 -> Decider(m,c0,c1,union({a},c2))
  []
  card(c0) >= THRESHOLD(m) & decision.m.0 -> Decider(m,c0,c1,c2)
  []
  card(c1) >= THRESHOLD(m) & decision.m.1 -> Decider(m,c0,c1,c2)
  []
  card(c2) >= THRESHOLD(m) & decision.m.2 -> Decider(m,c0,c1,c2)
```

The behaviour of agents described so far sets out how they make local decisions but they do not define how machine-level decisions are made. The `Decider` process is in charge of those. This centralised process collects local decisions made by the agents of a machine, offering a machine-level decision as soon as enough local decisions are gathered. This process is an abstraction that is useful for conciseness in specifying the behaviour of the machines but also for the sake of tractability. In a practical implementation of this protocol, each agent would implement the behaviour of the `Decider` process. Process `System` runs machines G and H with their respective `Decider` processes.

```
System = Nodes [|{|decide|}|]
    (Decider(g,{},{},{}) ||| Decider(h,{},{},{}))
```

We want to ensure that the system is safe – i.e. it must stick with one decision value once a decision is made—and live—i.e. it must offer a decision event before it is allowed to deadlock. Our discussion here is related to that on Sect. 4. In that section, we discussed how we can use a type of stochastic reasoning to choose the size of the set of pickets that is necessary to achieve a given number of good and malign agents, given some parameters for our stochastic model. In our CSP model, we talk about decision sets assuming that the pickets-set size and number of good and malign agents has been fixed, namely, the stochastic reasoning has

already been used to find these numbers. So, we limit ourselves to discuss the size of decision sets that is necessary to achieve safety and liveness.

Safety is ensured by setting a threshold that requires the participation of more than half of the benign nodes, namely, for machine `m`, `THRESHOLD(m)` $\geq$ `GOOD(m)`$/2$ + `BAD(m)` + 1, where `GOOD(m)`$/2$ is truncated integer division—we require the number of agents in each machine to be at least 2. If this threshold is set, the machine cannot decide on two different values on the same run of the protocol. Assume that agents in G supported two values, say `0` and `1`, then there must be `THRESHOLD(g)` many agents supporting either. That implies the existence of a benign agent that has supported two values, a possibility that our protocol does not allow; a contradiction. The same reasoning holds for H's independent decision. The requirement that H must decide on G's committed values, if one exists, ensures that if they both come to a decision, their value must match. As G can only commit to one value, by the same counting argument as before, H must decide on the same value as G.

Another assumption is required to ensure liveness. We expect H to come up with a decision if G fails to do so, but the agents in H may disagree on a decision value in the case they are left to independently select it. On a realistic implementation, agents will probably need to iterate if they fail to agree on a value within G or H until they eventually converge to a sufficiently agreed choice. How they achieve this is a separate topic but will likely involve coordinating input data and computing deterministically. For the sake of conciseness and tractability, we do not implement this process and we force enough benign agents in H to choose a common value (i.e. converge immediately), ensuring H comes to a decision. This immediate convergence is implemented by the `Convergence` process, which forces benign agents {`0..CN`} in machine H to choose the value `CV`—`CN` and `CV` are variables that parameterise our model. The *convergent system* is given by process `CSystem`. To achieve liveness, we need `GOOD(m)` $\geq$ `THRESHOLD(m)`—i.e., no malign agents are required to take part in the consensus—and that at least `THRESHOLD(m)`-many benign agents converge to the right value. From this inequality and the safety inequality before, one can derive the traditional lower bound on number of agents necessary for Byzantine agreement: $N = 3f + 1$ where N is the number of agents amongst whom f are malign.

```
AlphaConvergence = {| value.h.n | n <- {0..CN-1} |}

ConvergenceAux(0) = STOP
ConvergenceAux(i) = value.h.(i-1).CV -> ConvergenceAux(i-1)
Convergence = ConvergenceAux(CN)

CSystem = System [|AlphaConvergence|] Convergence
```

Similarly to what we did for the abstract model, we use the following refinement expression to capture these properties. The specification process `Spec` ensures that once a decision is made, only events deciding on that value are allowed be performed and that a decision event is offered initially—it can deadlock after a decision event is performed. Process `DSystem` captures a projection

of CSystem's behaviour onto decision events. We have used FDR to validate some instances of our model where thresholds are set in a way to ensure safety and liveness as discussed. We have also tested instances with insufficient thresholds to demonstrate how the model breaks down under those.

```
Spec = |~| m : MACHINES, v : VALUES @
    decision.m.v -> CHAOS({decision.mm.v | mm <- MACHINES})

DSystem = CSystem \ diff(Events,{|decision|})

assert Spec [F= DSystem
```

Interestingly, the inequality required to achieve safety alone does not restrict the proportion of benign agents that take part in the protocol. If we have, for instance, a single benign agent in a machine, a threshold requiring unanimity for decisions would still ensure safety. On the other hand, when both the safety and liveness inequalities are required, $> 2/3$ of agents must be benign. Thus, as machine G only needs to be safe, it can rely on there being as few as a single benign agent, whereas machine H, which must be safe and live, is required to have $> 2/3$ benign agents. Based on our stochastic calculations, for a fixed probability of an agent being malign, the number of agents that need to be selected to get a sample including at least one benign agent should be, in general, much smaller than the number needed for a sample including $> 2/3$ benign agents. Therefore, the number of agents required to implement G should be, in general, much smaller than the agents required to implement H. This fact supports our claim that G should be faster at coming to a decision when compared to H, given the smaller number of agents that are required to interact.

In many cases the "pickets" making up the back-up machine H will be entire qualified population of block creators, rather than being randomly chosen. In this case the hierarchical machine will precisely be the optimistic mechanism G backed up by classic Byzantine agreement. Moreover, typically, the 4 locations used by our protocol will be implemented in a distributed way by the agents involved. The correctness will depend on the signature mechanisms the blockchain has in place and also forms of the gossiping assumptions described earlier.

6 Related Work

Many classical protocols [17,18,20,25] exist to solve the Byzantine agreement problem [24]. The emergence of blockchains renewed the research community's interest in this problem—and more generally on the problem of achieving consensus in distributed systems—leading to a number of new protocols [1,3,4,6,12,15,22,30,32].

The first consensus protocol proposed for blockchains was *Proof-of-Work* (PoW) in the context of Bitcoin [22]. Intuitively speaking, in this protocol, *miners* (i.e. block producer candidates) attempt to solve a cryptographic puzzle, and

the first one who solves it is entitled to propose the next block to be added to the chain. Arguably, the main drawback of PoW protocols is how energy inefficient they can be [19,31]; the larger the network the more computing power is used to constantly solve these cryptographic puzzles. *Proof-of-Stake* (PoS) protocols have been proposed [1,5,12,15] as energy efficient alternatives to PoW ones. Hybrid PoW-PoS protocols have also been proposed [16].

In Proof-of-Stake protocols, agents signal their intention to participate in the block production process by *staking* a sum of cryptocurrency, i.e. the *stake*, they own. Staking means that this sum is locked (i.e. escrowed) for the duration of this process and it may be *slashed* as a means to punish malign behaviour. The frequency upon which agents are selected to participate in this process is proportional to the size of the stake. Note how in PoW computing power determines how often an agent is "selected" to produce a block as opposed to staked cryptocurrency in PoS. In PoS protocols, agents can be selected as a block producer but also as a member of a committee which is typically in charge of either electing block producers or *finalising blocks*, namely, determining whether a block is immutable and the only valid block at a given height. Before a block is deemed final, a number of candidate blocks at a given height might be "competing" to become final. Some PoS protocols rely on probabilistic mechanisms to determine the finality of a block—e.g. Algorand [12], Ouroboros [15]—whereas some others rely on deterministic mechanisms—e.g. Internet Consensus Computer [6], Casper FFG [5], Tendermint [3]. Our handover protocol is meant to be used as a part of a protocol to achieve deterministic finality, with our primary motivation being PoS. A PoS-based selection mechanism can be used to choose committees of agents—their sizes are based on our stochastic calculations—to implement machines G and H and to decide on the next *final* block using the handover protocol.

Despite being designed to be part of a fully-fledged blockchain consensus protocol, the handover protocol alone is closer in nature to mutual exclusion, though adapted for linking agreement protocols like Byzantine agreement [17, 20,23,32]. Abstractly speaking, these protocols have been designed around the use of the votes to form decisions and of a threshold/quorum to ensure safety. In fact, the PBFT (Practical Byzantine Fault Tolerance) protocol [20] specifically—and this voting mechanism more generally—has been a source of inspiration for many current blockchain protocols, including ours.

7 Conclusions

In this paper we have used formal tools to understand how consensus can arise in decentralised systems. Essentially we have set out a programmatic approach to laying down and analysing consensus: given a population of potential block creators and the potentially multiple perspectives of different users we need to establish a trust model that they are all happy with. We then have the job of having the blockchain select sufficient groups of pickets and decision criteria that all can be sure of any positive decisions they make.

On the assumption that we can incentivise most malign participants to participate apparently properly, this will give us all we need. But an essential part of such motivation is that the malign know that if they do not collaborate like this they will be defeated by a back up mechanism.

We have shown how to formalise both the primary and secondary mechanisms as Unitary Consensus Machines. While much of our treatment was inspired by process algebra, we were able to both design and verify the crucial protocol that links a hierarchy of decision making in CSP and FDR.

By allowing such hierarchical consensus decisions, we believe that we have tools for making blockchains more varied and flexible. We hope that our approach to creating the component machines which compose together to provide consensus can be automated.

It is only natural - to people steeped in such languages and tools - that CSP coupled with FDR is a good way to model complex interactions in decentralised consensus. We are pleased to have demonstrated the truth of this intuition. While the full systems representing consensus may be too involved to fit within the abstractions of such tools, it is comforting that like so many other areas of concurrent reasoning, we can find levels where they bring real benefit. We have modelled other aspects of blockchain using CSP and FDR.

We hope that others will be found, and that our tools for bringing clarity to the topic of consensus will find many interesting applications.

References

1. Bentov, I., Gabizon, A., Mizrahi, A.: Cryptocurrencies without proof of work. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 142–157. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_10
2. Brookes, S.D., Roscoe, A.W.: CSP: A Practical Process Algebra, 1 edn., pp. 187–222. Association for Computing Machinery, New York (2021)
3. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. CoRR abs/1807.04938 (2018). http://arxiv.org/abs/1807.04938
4. Buterin, V.: Ethereum: a next-generation smart contract and decentralized application Platform (2014). https://ethereum.org/whitepaper/
5. Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR abs/1710.09437 (2017). http://arxiv.org/abs/1710.09437
6. Camenisch, J., Drijvers, M., Hanke, T., Pignolet, Y.A., Shoup, V., Williams, D.: Internet computer consensus. In: Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing, PODC 2022, pp. 81–91. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3519270.3538430
7. Dowd, K., Cotter, J., Humphrey, C., Woods, M.: How unlucky is 25-sigma? J. Portfolio Manag. **34**, 76–80 (2008). https://doi.org/10.3905/jpm.2008.709984
8. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM (JACM) **35**(2), 288–323 (1988)
9. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)

10. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13
11. Gibson-Robinson, T., Lowe, G.: Symmetry reduction in CSP model checking. Int. J. Softw. Tools Technol. Transfer **21**(5), 567–605 (2019). https://doi.org/10.1007/s10009-019-00516-4
12. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, pp. 51–68. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3132747.3132757
13. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
14. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice Hall (1985)
15. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 357–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_12
16. King, S., Nadal, S.: Ppcoin: peer-to-peer crypto-currency with proof-of-stake. Self-published paper, 19 August 2012
17. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229
18. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982). https://doi.org/10.1145/357172.357176
19. Li, X., Zhu, Q., Qi, N., Huang, J., Yuan, Y., Wang, F.Y.: Blockchain consensus algorithms: a survey. In: 2021 China Automation Congress (CAC), pp. 4053–4058 (2021). https://doi.org/10.1109/CAC53003.2021.9728000
20. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383
21. Lowe, G.: Casper: a compiler for the analysis of security protocols. J. Comput. Secur. **6**(1–2), 53–84 (1998)
22. Nakamoto, S., et al.: Bitcoin: a peer-to-peer electronic cash system (2008)
23. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC 2014, pp. 305–320. USENIX Association (2014)
24. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980). https://doi.org/10.1145/322186.322188
25. Rabin, M.O.: Randomized byzantine generals. In: 24th Annual Symposium on Foundations of Computer Science (SFCS 1983), pp. 403–409 (1983). https://doi.org/10.1109/SFCS.1983.48
26. Roscoe, A.W.: Model-checking CSP. In: International Series in Computer Science. Prentice Hall (1994). http://www.cs.ox.ac.uk/people/bill.roscoe/publications/50.ps
27. Roscoe, A.W.: The Theory and Practice of Concurrency. Series in Computer Science. Prentice Hall (1998)
28. Roscoe, A.W.: Understanding Concurrent Systems. Springer, London (2010). https://doi.org/10.1007/978-1-84882-258-0

29. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_7
30. Wood, G.: Ethereum yellow paper. https://ethereum.github.io/yellowpaper/paper.pdf
31. Xiao, Y., Zhang, N., Lou, W., Hou, Y.T.: A survey of distributed consensus protocols for blockchain networks. IEEE Commun. Surv. Tutor. **22**(2), 1432–1465 (2020). https://doi.org/10.1109/COMST.2020.2969706
32. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, pp. 347–356. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3293611.3331591