Frosty: Bringing strong liveness guarantees to the Snow family of consensus protocols.

AARON BUCHWALD, Ava Labs, USA
STEPHEN BUTTOLPH, Ava Labs, USA
ANDREW LEWIS-PYE, London School of Economics, UK
PATRICK O'GRADY, Ava Labs, USA
KEVIN SEKNIQI, Ava Labs, USA

Snowman is the consensus protocol implemented by the Avalanche blockchain and is part of the Snow family of protocols, first introduced through the original Avalanche leaderless consensus protocol [28]. A major advantage of Snowman is that each consensus decision only requires an expected constant communication overhead per processor in the 'common' case that the protocol is not under substantial Byzantine attack, i.e. it provides a solution to the scalability problem which ensures that the expected communication overhead per processor is independent of the total number of processors n during normal operation. This is the key property that would enable a consensus protocol to scale to 10,000 or more independent validators (i.e. processors). On the other hand, the two following concerns have remained:

(1) Providing formal proofs of consistency for Snowman has presented a formidable challenge.
(2) Liveness attacks exist in the case that a Byzantine adversary controls more than $O(\sqrt{n})$ processors, slowing termination to more than a logarithmic number of steps.

In this paper, we address the two issues above. We consider a Byzantine adversary that controls at most $f < n/5$ processors. First, we provide a simple proof of consistency for Snowman. Then we supplement Snowman with a 'liveness module' that can be triggered in the case that a substantial adversary launches a liveness attack, and which guarantees liveness in this event by *temporarily* forgoing the communication complexity advantages of Snowman, but without sacrificing these low communication complexity advantages during normal operation.

1 INTRODUCTION

Recent years have seen substantial interest in developing consensus protocols that work efficiently at scale. In concrete terms, this means looking to minimize the latency and communication complexity per consensus decision as a function of the number of processors (participants/validators) n. The Dolev-Reischuk bound [14], which asserts that deterministic protocols require $O(n^2)$ communication complexity per consensus decision, presents a fundamental barrier in this regard: deterministic protocols that can tolerate a Byzantine (i.e. arbitrary) adversary of size $O(n)$ must necessarily suffer a quadratic blow-up in communication cost as the size of the network grows. It is precisely this relationship that makes these protocols susceptible to considerable slowdown when a high number of processors is present.

Probabilistic sortition. One approach to dealing with this quadratic blow-up in communication cost, as employed by protocols such as Algorand [10], is to utilize probabilistic *sortition* [1, 19]. Rather than have *all* processors participate in every consensus decision, the basic idea is to sample a *committee* of sufficient size that the proportion of Byzantine committee members is almost certainly close to the proportion of all processors that are Byzantine. Sampled committees of constant bounded size can then be used to implement consensus, thereby limiting the communication cost. In practical terms, however, avoiding Byzantine control of committees requires each committee to have a number of members sufficient that the *quadratic communication cost for the committee* is already substantial, e.g. Algorand requires committees with k members, where k is of the order of one thousand, meaning that k^2 is already large.

The Snow family of consensus protocols. In [28], a family of consensus protocols was specified, providing an alternative approach to limiting communication costs. These protocols are all based on a common approach that is best described by considering a binary decision game. For the sake of simplicity, let us initially consider the Snowflake protocol[1], which uses three parameters: k, $\alpha > k/2$, and β (for the sake of concreteness, in this paper we will focus on the example that $k = 80$). Suppose that each processor begins with an initial color, either red or blue. Each processor p then proceeds in rounds. In each round, p randomly samples k processors from the total population and asks those processors to report their present color. If at least α of the reported values are the opposite of p's present color, then p adopts that opposite color. If p sees β consecutive rounds in which at least α of the reported values are red, then p decides red (and similarly for blue).

The outcome of this dynamic sampling process can be informally described as follows when the adversary is sufficiently bounded (a formal analysis for a variant of Snowflake that we call Snowflake$^+$ is given in Section 4). Once the proportion of the population who are red, say, passes a certain tipping point, it holds with high probability that the remainder of the (non-Byzantine) population will quickly become red (and symmetrically so for blue). If β is set appropriately, then the chance that any correct processor decides on red before this tipping point is reached can be made negligible, meaning that once any correct processor decides on red (or blue), they can be sure that all other correct processors will quickly decide the same way. The chance that correct processors decide differently can thus be made negligible through an appropriate choice of parameter values. If correct processors begin heavily weighted in favor of one color, then convergence on a decision value will happen very quickly, while variance in random sampling is required to tip the population in one direction in the case that initial inputs are evenly distributed.

While the discussion above considers a single binary decision game, the 'Snowman' protocol, formally described and analysed for the first time in this paper, shows that similar techniques can be used to efficiently solve State Machine Replication (SMR) [30]. The transition from simple consensus (Byzantine Agreement [20]) to an efficient SMR protocol is non-trivial, and is described in detail in Sections 5 and 6. A major benefit of the approach is that it avoids the need for all-to-all communication. In an analysis establishing that there is only a small chance of consistency failure, the value of k can be specified independent of n, and each round requires each processor to collect reported values from only k others.

Our contribution. The Snowman protocol is presently used by the Avalanche blockchain to implement SMR. However, the two following concerns have remained:

(1) Providing formal proofs of consistency for Snowman has presented a formidable challenge.
(2) Liveness attacks exist in the case that a Byzantine adversary controls more than $O(\sqrt{n})$ processors [28], meaning that finalization is no longer guaranteed to occur in a logarithmic number of steps.

In this paper, we consider a Byzantine adversary that controls at most $f < n/5$ processors, and address the two issues above. With respect to issue (1):

- We describe a variant of Snowflake, called Snowflake$^+$.
- For appropriate choices of parameter values, we give a simple proof that Snowflake$^+$ satisfies 'validity' and 'agreement' except with small error probability.
- We give a complete specification of a version of Snowman that builds on Snowflake$^+$. This is the first formal description of the Snowman protocol.

[1]In [28], other variants such as the Slush and Snowball protocols are also described.

- For appropriate choices of parameter values, we give a simple proof that the resulting Snowman protocol satisfies consistency except with small error probability.
- We also describe a variant of Snowflake$^+$ called Error-driven Snowflake$^+$, that can be used to give very low latency in the 'common case'.

With regard to issue (2), we note that malicious liveness attacks on Avalanche have not been observed to date. It is certainly desirable, however, to have strong guarantees in the case that a large adversary launches an attack on liveness. The approach we take in this paper is therefore to strike a practical balance. More specifically, we aim to specify a protocol that is optimised to work efficiently in the 'common case' that there is no substantial Byzantine attacker, but which also provides a 'fallback' mechanism in the worst case of a substantial attack on liveness. To this end, we then describe how to supplement Snowman with a 'liveness module'. The basic idea is that one can use Snowman to reach fast consensus under normal operation, and can then trigger an 'epoch change' that temporarily implements some standard quorum-based protocol to achieve liveness in the case that a substantial adversary attacks liveness. In the (presumably rare) event that a substantial adversary attacks liveness, liveness is thus ensured by *temporarily* forgoing the communication complexity advantages of Snowman during normal operation. The difficulty in implementing such a module is to ensure that interactions between the different modes of operation do not impact consistency. We give a formal proof that the resulting protocol, called Frosty, is consistent and live, except with small error probability. To the best of our knowledge, this approach is novel and introduces a new spectrum of optimizations for consensus protocols.

Paper structure. Section 2 describes the formal model. Section 3 describes Snowflake$^+$ and gives pseudocode for the protocol. Section 4 gives a simple proof of agreement and validity for Snowflake$^+$ and describes Error-driven Snowflake$^+$. Section 5 describes the Snowman protocol, including pseudocode. Section 6 gives a simple proof of consistency for Snowman. Section 7 describes the liveness module and gives pseudocode for the resulting protocol, called Frosty. Section 8 proves liveness and consistency for Frosty.

2 THE MODEL

We consider a set $\Pi = \{p_0, \dots, p_{n-1}\}$ of n processors. Processor p_i is told i as part of its input. For the sake of simplicity, we assume a static adversary that controls up to f of the processors, where f is a known bound. Generally, we will assume $f < n/5$ (the bound $f < n/5$ is chosen only so as to give as simple a proof as possible in Section 4, and providing an analysis for larger f is the subject of future work). A processor that is controlled by the adversary is referred to as *Byzantine*, while processors that are not Byzantine are *correct*. Byzantine processors may display arbitrary behaviour, modulo our cryptographic assumptions (described below).

Cryptographic assumptions. Our cryptographic assumptions are standard for papers in distributed computing. Processors communicate by point-to-point authenticated channels. We use a cryptographic signature scheme, a public key infrastructure (PKI) to validate signatures, and a collision-resistant hash function H. We assume a computationally bounded adversary. Following a common standard in distributed computing and for simplicity of presentation (to avoid the analysis of certain negligible error probabilities), we assume these cryptographic schemes are perfect, i.e. we restrict attention to executions in which the adversary is unable to break these cryptographic schemes. In a given execution of the protocol, hash values are thus assumed to be unique.

Communication. As noted above, processors communicate using point-to-point authenticated channels. We consider the standard synchronous model: for some known bound Δ, a message sent at time t must arrive by time $t + \Delta$.

The binomial distribution. Consider k independent and identically distributed random variables, each of which has probability x of taking the value 'red'. We let $\mathrm{Bin}(k, x, m)$ denote the probability that m of the k values are red, we write $\mathrm{Bin}(k, x, \geq m)$ to denote the probability that *at least* m values are red (and similarly for $\mathrm{Bin}(k, x, \leq m)$).

Dealing with small probabilities. In analysing the security of a cryptographic protocol, one standardly regards a function $f : \mathbb{N} \rightarrow \mathbb{N}$ as *negligible* if, for every $c \in \mathbb{N}$, there exists $N_c \in \mathbb{N}$ such that, for all $x \geq N_c$, $|f(x)| < 1/x^c$. Our concerns here, however, are somewhat different. As noted above, we assume the cryptographic schemes utilized by our protocols are perfect. For certain *fixed* parameter values (e.g. setting $n = 500$, $k = 80$, $\alpha = 41$ and $\beta = 12$ in an instance of Snowflake, as described in Section 1), we want to be able to argue that error probabilities are sufficiently small that they can reasonably be dismissed.

In our analysis, we will therefore identify certain events as occurring with *small* probability (e.g. with probability $< 10^{-20}$), and may then condition on those events not occurring. If $\mathbb{P}(A)$ is small and $\mathbb{P}(B)$ is small, then we will be happy to assume that $\mathbb{P}(A|\neg B)$ is small, while taking care to ensure this principle is not abused. Often, we will consider specific events, such as the probability in a round-based protocol that a given processor performs a certain action x in a given round. In dismissing small error probabilities, one then has to take account of the fact that there may be many opportunities for an event of a given type to occur, e.g. any given processor may perform action x in any given round. How reasonable it is to condition on no correct processor performing action x may therefore depend on the number of processors and the number of rounds, and we assume 'reasonable' bounds on these values. As an example, consider the Snowflake protocol, as described in Section 1, and suppose $k = 80$ and that at most 1/5 of the processors are Byzantine. Suppose that, at the beginning of a certain round, at least 75% of the correct processors are red. Then a simple calculation for the binomial distribution shows that the probability a correct processor receives at least 72 blue responses from the 80 processors it samples in that round is upper bounded by 1.18×10^{-20},i.e. $\mathrm{Bin}(80, 0.2 + (0.8 \times 0.25), \geq 72) < 1.18 \times 10^{-20}$. To upper bound the probability that there exists *any* round in which at least 75% of correct processors are red and *some* correct processor receives at least 72 blue responses, we just apply the union bound. For the sake of concreteness, suppose that at most 10,000 processors run the protocol for at most 1000 years, executing at most 5 rounds a second. This means that less than 1.6×10^{11} rounds are executed. Since there are at most 10,000 processors, the union bound thus gives a cumulative error probability less than 2×10^{-5}. We will address such accountancy issues as they arise.

We stress that accounting for *small* error probabilities in the manner described above (rather than showing error probabilities are negligible functions of the parameter inputs) also allows us to give particularly straightforward security proofs for Snowflake$^+$, Snowman, and Frosty.

A comment on the use of synchrony. We take an approach similar to that in the 'Bitcoin Backbone' paper [16], and simplify our analysis by having correct processors execute the protocol executions in cleanly defined *rounds*. Each correct processor thus samples the values of some others in round 1, before adjusting local values based on that sample. All correct processors then proceed to round 2, and so on. In Section 10, we discuss how future work may augment this analysis to deal with *responsive* [25] versions of the protocols described here.

3 A SIMPLE PROTOCOL FOR BYZANTINE AGREEMENT: SNOWFLAKE$^+$.

We begin by describing a simple probabilisitic protocol for binary Byzantine Agreement, called Snowflake$^+$, which will act as a basic building block for the Snowman protocol (described later in Section 5).

The inputs. Each processor p_i begins with a value $\texttt{input}_i \in \{0, 1\}$.

The requirements. A probabilistic protocol for Byzantine agreement is required to satisfy the following properties, except with small error probability:
Agreement: No two correct processors output different values.
Validity: If every correct processor i has the same value $\texttt{input}_i$, then no correct processor outputs a value different than this common input.
Termination: Every correct processor gives an output.

Recalling Snowflake. Since Snowflake$^+$ is a simple variant of Snowflake, let us first informally recall the Snowflake protocol. Snowflake uses three parameters: k, $\alpha > k/2$, and β. Each processor p_i maintains a variable $\texttt{val}_i$, initially set to $\texttt{input}_i$. The instructions proceed in rounds. In each round, processor p_i randomly samples k processors from the total population and asks each of those processors p_j to report their present value $\texttt{val}_j$. If at least α of the reported values are the opposite of p_i's present value $\texttt{val}_i$, then p_i sets $\texttt{val}_i := 1 - \texttt{val}_i$. If p_i sees β consecutive rounds in which at least α of the reported values are 1, then p_i decides 1 (and similarly for 0).

Snowflake$^+$ is similar to Snowflake, except that we now use two parameters α_1 and α_2, rather than a single parameter α.

The protocol parameters for Snowflake$^+$. The protocol parameters are $k, \alpha_1, \alpha_2, \beta \in \mathbb{N}_{>0}$ and satisfy the constraints that $\alpha_1 > k/2$ and $\alpha_2 \geq \alpha_1$. Each processor p_i also maintains a variable $\texttt{val}_i$, initially set to $\texttt{input}_i$. The parameter k determines sample sizes. The parameter α_1 is used to determine when processor p_i changes $\texttt{val}_i$. Parameters α_2 and β are used to determine the conditions under which p_i will output and terminate.

The protocol instructions for Snowflake$^+$. The instructions are divided into rounds, with round s occurring at time $2\Delta s$. In round s, processor p_i:

(1) Sets $\langle p_{1,s}, \dots p_{k,s} \rangle$ to be a sequence of k processors (specific to p_i). For $j \in [1, k]$, $p_{j,s}$ is sampled from the uniform distribution[2] on all processors (so sampling is "with replacement").
(2) Requests each $p_{j,s}$ (for $j \in [1, k]$) to report its present value $\texttt{val}_j$.
(3) Waits time Δ and reports its present value $\texttt{val}_i$ to any processor that has requested it in round s.
(4) Waits another Δ and considers the values reported in round s:
- If at least α_1 of the reported values are $1 - \texttt{val}_i$, then p_i sets $\texttt{val}_i := 1 - \texttt{val}_i$.
- If p_i has seen β consecutive rounds in which at least α_2 of the reported values are equal to $\texttt{val}_i$, then p_i outputs this value and terminates.

The pseudocode is described in Algorithm 1.

In Section 4, we will show that Snowflake$^+$ satisfies agreement and validity for appropriate choices of the protocol parameters, and so long as $f < n/5$. We do not give a formal analysis of termination for Snowflake$^+$: Once Snowflake$^+$ has been used to define Snowman in Section 5, in Section 7 we

[2]In proof-of-stake implementations, sampling will be stake-weighted, but, for the sake of simplicity of presentation, we ignore such issues here.

will describe how to augment Snowman with a liveness module (guaranteeing termination), which is formally analysed in Section 8.

Algorithm 1 Snowflake$^+$: The instructions for processor p_i

Inputs
$\texttt{input}_i \in \{0,1\}$ ▷ p_i's input
$\Delta, k, \alpha_1, \alpha_2, \beta \in \mathbb{N}$ ▷ Protocol parameters
Local variables
$\texttt{val}_i$, initially set to $\texttt{input}_i$ ▷ p_i's present 'value'
$\texttt{count}$, initially set to 0 ▷ Output once $\texttt{count}$ reaches β
$v_i(j,s)$, initially undefined ▷ Stores at most one received value per round

The instructions for round s, beginning at time $2\Delta s$:
    Form sample sequence $\langle p_{1,s}, \dots p_{k,s} \rangle$; ▷ Sample with replacement
    For $j \in [1,k]$, send s to $p_{j,s}$; ▷ Ask $p_{j,s}$ for present value
    Wait Δ;
    For each j such that p_i has received s from p_j:
        Send $(s, \texttt{val}_i)$ to p_j;
    Wait Δ;
    For each $j \in [1,k]$:
        If p_i has received a first message (s,v) from $p_{j,s}$;
            Set $v_i(j,s) := v$;
        Else set $v_i(j,s) := \bot$;
    If $|\{j : 1 \le j \le k, v_i(j,s) == 1 - \texttt{val}_i\}| \ge \alpha_1$, set $\texttt{val}_i := 1 - \texttt{val}_i$, $\texttt{count} := 0$;
    If $|\{j : 1 \le j \le k, v_i(j,s) == \texttt{val}_i\}| < \alpha_2$, set $\texttt{count} := 0$;
    If $|\{j : 1 \le j \le k, v_i(j,s) == \texttt{val}_i\}| \ge \alpha_2$, set $\texttt{count} := \texttt{count} + 1$;
    If $\texttt{count} \ge \beta$, output $\texttt{val}_i$ and terminate.

4 SECURITY ANALYSIS OF SNOWFLAKE$^+$

We assume $f < n/5$. For the sake of concreteness, we establish satisfaction of agreement and validity for $k = 80$, $\alpha_1 = 41$, $\alpha_2 = 72$, and $\beta = 12$, under the assumption that the population size $n \ge 500$. We make the assumption that $n \ge 500$ only so as to be able to give as simple a proof as possible: a more fine-grained analysis for smaller n is the subject of future work.

Coloring the processors. Since 0 and 1 are not generally used as adjectives, let us say a correct processor p_i is 'blue' in round s if $\texttt{val}_i = 0$ at the beginning of round s, and that p_i is 'red' in round s if $\texttt{val}_i = 1$ at the beginning of round s. Recall (from Algorithm 1) that $v_i(j,s)$ is the color that p_j reports to p_i in round s. We'll say a correct processor p_i 'samples x blue' in round s if $|\{j : 1 \le j \le k, v_i(j,s) = 0\}| = x$ (and similarly for red). We'll also extend this terminology in the obvious way, by saying that a processor outputs 'blue' if it outputs 0 and outputs 'red' if it outputs 1. In the below, we'll focus on the case that, in the first round in which a correct processor outputs (should such a round exist), some correct processor outputs red. A symmetric argument can be made for blue.

In the following argument, we will adopt the conventions described in Section 2 concerning the treatment of small error probabilities.

Establishing Agreement. First, let us consider what happens when the proportion of correct processors that are red reaches a certain threshold. In particular, let us consider what happens when at least 75% of the correct processors are red in a given round s. A simple calculation for the binomial distribution shows that the probability a given correct processor is red in round $s+1$ is then at least 0.9555, i.e. $\text{Bin}(80, 0.8 \times 0.75, \geq 41) > 0.9555$. Assuming a population of at least 500, of which at least 80% are correct (meaning that at least 400 are correct), another simple calculation for the binomial distribution shows that the probability that it fails to be the case that more than 5/6 of the correct processors are red in round $s+1$ is upper bounded by 1.59×10^{-20}, i.e. $\text{Bin}(n, 0.9555, \leq 5n/6) < 1.59 \times 10^{-20}$ for $n \geq 400$. For a population of at least 500, and by iterating this argument over rounds, we can therefore condition on the following event:

($\dagger_1$) If at least 75% of the correct processors are red in any round s, then, in all rounds s' with $s' > s$, more than 5/6 of the correct processors are red.

A simple calculation for the binomial distribution shows that if at least 75% of correct processors are red in a given round s, then the probability that a given correct processor p_i samples at least 72 blue in round s is upper bounded by 1.18×10^{-20}, i.e. $\text{Bin}(80, 0.2 + (0.8 \times 0.25), \geq 72) < 1.18 \times 10^{-20}$. We can therefore condition on the following event:

($\dagger_2$) If at least 75% of the correct processors are red in any round s, then no correct processor samples at least 72 blue in round s.

Another simple calculation for the binomial distribution shows that, if *at most* 75% of correct processors are red in a given round s, then the probability a given correct processor samples 72 or more red in round s is upper bounded by 0.0131, i.e. $\text{Bin}(80, (0.75 \times 0.8) + 0.2, \geq 72) < 0.0131$. So, if we consider 12 consecutive rounds in which at most 75% of the correct processors are red, the probability a given correct processor samples at least 72 red in each round is upper bounded by 10^{-22}. We can therefore condition on the following event:

($\dagger_3$) If a correct processor outputs red in round $s+11$, then, for at least one round $s' \in [s, s+11]$, at least 75% of correct nodes are red in round s'.

According to ($\dagger_3$), if a correct processor is the (potentially joint) first to output and outputs red after sampling in round $s+11$, at least one round $s' \in [s, s+11]$ must satisfy the condition that at least 75% of correct processors are red in round s'. Then, given our prior assumption ($\dagger_1$), at least 5/6 of the correct processors must be red in all rounds $> s'$. From assumption ($\dagger_2$), it follows that no correct processor ever outputs blue. This suffices to show that Agreement is satisfied, except with small error probability.

Establishing Validity. A similar (but even simpler) argument suffices to establish validity. Suppose that all honest nodes have the same input, red say (i.e. 1). By the same reasoning as above, since round 0 satisfies the condition that at least 75% (in fact 100%) of correct processors are red, we can then condition on the following event:

($\dagger_4$) In every round, more than 5/6 of the correct processors are red.

From ($\dagger_2$) and ($\dagger_4$) it follows that no correct processor outputs blue, as required.

Dealing with different parameter values. The argument above is easily adapted to deal with alternative parameter values. If we fix $\alpha_1 := \lfloor k/2 \rfloor + 1$, then error probabilities will be smaller for larger values of α_2 and β. For smaller values of α_2, similar error probabilities can be obtained by increasing β – the required values for β are easily found by adapting the binomial calculations above. Examples are given in Section 4.1.

Counting the accumulation of small error probabilities. Suppose that at most 10,000 processors run the protocol for at most 1000 years, executing at most five rounds each second.

In the analysis above, we considered the case that there is a first round s in which at least 75% of the correct processors are red. We concluded that the probability that it fails to be the case that more than 5/6 of the correct processors are red in round $s+1$ is upper bounded by 1.59×10^{-20}. We then iterated this argument over rounds. If there are five rounds per second then, over a period of 1000 years, this means that less than 1.6×10^{11} rounds are executed. The union bound thus gives a cumulative error probability of less than 3×10^{-9}.

We also considered the case that at least 75% of correct processors are red in a given round s, and concluded that the probability that a given correct processor p_i samples at least 72 blue in round s is upper bounded by 1.18×10^{-20}. For 10,000 processors executing at most 1.6×10^{11} rounds, the union bound thus gives a cumulative error probability of less than 2×10^{-5}.

We also considered any 12 consecutive rounds in which at most 75% of the correct processors are red, and concluded that the probability a given correct processor samples at least 72 red in each round is upper bounded by 10^{-22}. For 10,000 processors executing at most 1.6×10^{11} rounds, this gives a cumulative error probability of less than 2×10^{-7}.

Adding these cumulative error probabilities (and applying the union bound), we conclude that, for at most 10,000 processors running the protocol for 1000 years, the chance of a consistency violation is upper bounded by 3×10^{-5}.

4.1 Error-driven Snowflake$^+$

In Section 4, we considered a fixed value $\alpha_2 = 72$, for $k = 80$. While considering a fixed α_2 suffices for the analysis there, it is also useful to consider multiple values of α_2, giving rise to a number of different conditions for termination. Considering a range of termination conditions for different values of α_2 serves two functions: Considering lower values of α_2 allows one to deal with a greater percentage of offline/faulty processors, while higher values of α_2 give quick decision conditions and low latency in the good case.

Error-driven Snowflake$^+$ is the same as Snowflake$^+$, except that one simultaneously considers multiple possible values of $\alpha_2 \leq k$. Each α_2 now gives rise to a different β that determines the conditions for termination. The corresponding values are shown in Table 1.

How the values in Table 1 are calculated. In Section 4, it was ($\dagger_3$) which played a crucial role in establishing the relationship between α_2 and β for a given error probability $\epsilon > 0$. Assuming that at most 75% of correct processors are red, a simple calculation for the binomial distribution then upper bounds the probability p that a given correct processor samples at least α_2 red in a given round. For a given error probability ϵ, the corresponding β shown in Table 1 is the least integer such that $p^\beta < \epsilon$. The value p^β upper bounds the probability of a given correct processor sampling at least α_2 red in β given consecutive rounds, under the assumption that at most 75% of correct processors are red in each round.

Table 1 also shows how β depends on α_2 for larger error bounds ($\epsilon < 10^{-14}$ and $\epsilon < 10^{-6}$). Correct processors may use the corresponding lower values of β in the case that they are willing to accept higher error probabilities for the sake of achieving low latency, i.e. terminating in a small number of rounds.

Low latency in standard operation. Analysis of data from the Avalanche blockchain shows that, at any given point in time, one can expect close to 100% of contributing processors to act correctly. For Error-driven Snowflake$^+$, this corresponds to a scenario where the vast majority of processors are correct, and where initial inputs are generally highly biased in favor of one color. The conditions in Table 1 that allow for quick termination (using $\beta = 3, 4$ or 5, say) can therefore be

α_2	β for $\epsilon < 10^{-22}$	β for $\epsilon < 10^{-14}$	β for $\epsilon < 10^{-6}$
80	3	2	1
79	4	3	1
78	5	3	2
77	5	4	2
76	6	4	2
75	7	5	2
74	9	6	3
73	10	7	3
72	12	8	4
71	15	10	4
70	18	12	5
69	23	15	7
68	29	18	8
67	37	24	10
66	48	31	14
65	65	41	18

Table 1. The required β as a function of α_2 and the error bound.

expected to be commonly satisfied, and give a significant improvement in latency for the standard case that most processors act correctly.

The accumulation of error probabilities. Accepting multiple conditions for termination gives an overall error probability that can be (generously) upper bounded simply by applying the union bound. In Table 1, 16 different termination conditions are listed. If processors apply all of these termination conditions simultaneously, then this will lead to at most a 16-fold increase in error probability.[3]

5 THE SNOWMAN PROTOCOL

Since the Snowman protocol is not specified in the original whitepaper [28], we give a precise description and pseudocode here.

5.1 Transactions and blocks

To specify a protocol for State-Machine-Replication (SMR), we suppose processors are sent (signed) transactions during the protocol execution: Formally this can be modeled by having processors be sent transactions by an *environment*, e.g. as in [21]. Processors may use received transactions to form *blocks* of transactions. To make the analysis as general as possible, we decouple the process of

[3]While simple, this analysis significantly overestimates the error probability. If one applies all 16 termination conditions listed in Table 1, this does not impact the probability that $(\dagger_1)$ or $(\dagger_2)$ (from the analysis of Section 4) hold, and only impacts the probability that $(\dagger_3)$ holds. In Section 4, we concluded that if 10,000 processors run the protocol for 1000 years, executing at most 5 rounds a second, then the probability that $(\dagger_3)$ fails is less than 2×10^{-7}. Applying the union bound and multiplying by 16, we conclude that $(\dagger_3)$ fails to hold with probability at most 3.2×10^{-6} when the 16 different termination conditions of Table 1 are applied. Adding the probability that either of $(\dagger_1)$ or $(\dagger_2)$ fails to hold still gives an overall error probability of less than 3×10^{-5} (the same bound provided in Section 4).

block production from the core consensus engine. We therefore suppose that some given process for block generation operates in the background, and that valid blocks are gossiped throughout the network. We do not put constraints on the block generation process, and allow that it may produce equivocating blocks, etc. In practice, block generation could be specified simply by having a rotating sequence of leaders propose blocks, or through a protocol such as Snowman^{++}, as actually used by the present implementation of the Avalanche blockchain.

Blockchain structure. We consider a fixed *genesis block* b_0. In a departure from the approach described in the original Avalanche whitepaper [28], which built a directed acyclic graph (DAG) of blocks, we consider a standard blockchain architecture in which each block b other than b_0 specifies a unique *parent*. If b' is the parent of b, then b is referred to as a *child* of b'. In this case, the ancestors of b are b and any ancestors of b'. Every block must have b_0 as an ancestor. The descendants of any block b are b and any descendants of its children. The *height* of a block b is its number of ancestors other than b, meaning that the height of b_0 is 0. By a *chain* (ending in b_h), we mean a sequence of blocks $b_0 * b_1 * \cdots * b_h$, such that $b_{h'+1}$ is a child of $b_{h'}$ for $h' < h$.[4]

5.2 Overview of the Snowman protocol

To implement SMR, our approach is to run multiple instances of Snowflake^{+}. To keep things simple, consider first the task of reaching consensus on a block of height 1. Suppose that multiple children of b_0 are proposed over the course of the execution and that we must choose between them. To turn this decision problem into multiple binary decision problems, we consider the hash value $H(b_1)$ of each proposed block b_1 of height 1, and then run one instance of Snowflake^{+} to reach consensus on the first bit of the hash. Then we run a second instance to reach consensus on the second bit of the hash, and so on. Working above a block of any height h, the same process is then used to finalize a block of height $h + 1$. In this way, multiple instances of Snowflake^{+} are used to reach consensus on a chain of hash values $H(b_0) * H(b_1) * \dots$.

This process would not be efficient if each round required a separate set of correspondences for each instance of Snowflake^{+}, but this is not necessary. Just as in Snowflake^{+}, at the beginning of each round s, processor p_i samples a single sequence $\langle p_{1,s}, \dots p_{k,s}\rangle$ of k processors. Since we now wish to reach consensus on a sequence of blocks, each processor $p_{j,s}$ in the sample is now requested to report its presently preferred chain, rather than a single bit value. The first bit of the corresponding hash sequence is then used by p_i as the response of $p_{j,s}$ in a first instance of Snowflake^{+}. If this first bit agrees with p_i's resulting value in that instance of Snowflake^{+}, then the second bit is used as the value reported by $p_{j,s}$ in a second instance of Snowflake^{+}, and so on.

A note on some simplifications that are made for the sake of clarity of presentation. When a processor $p_{j,s}$ is requested by p_i to report its presently preferred chain (ending with b, say), we have $p_{j,s}$ simply send the given sequence of blocks. In reality, this would be very inefficient and the present implementation of Snowman deals with this by having $p_{j,s}$ send a hash of b instead. This potentially causes some complexities, because p_i may not have seen b (meaning that it does not necessarily know how to interpret the hash). This issue is easily dealt with, but it would be a distraction to go into the details here.

The variables, functions and procedures used by p_i. The protocol instructions make use of the following variables and functions (as well as others whose use should be clear from the pseudocode):

- b_0: The genesis block.

[4] Throughout this paper, '$*$' denotes concatenation.

- `blocks`: Stores blocks received by p_i (and verified as valid). Initially it contains only b_0, and it is automatically updated over time to include any block included in any message received or sent by p_i.
- $\texttt{val}(\sigma)$: For each finite binary string σ, $\texttt{val}(\sigma)$ records p_i's presently preferred value for the next bit of the chain of hash values $H(b_0) * H(b_1) * \dots$, should the latter extend σ.
- `pref`: The initial segment of the chain of hash values that p_i presently prefers. We write $|\texttt{pref}|$ to denote the length of this binary string.
- `final`: The initial segment of the chain of hash values that p_i presently regards as final.
- $\texttt{chain}(\sigma)$: If there exists a greatest $h \in \mathbb{N}$ such that $\sigma = H(b_0) * \cdots * H(b_h) * \tau$ for a chain of blocks $b_0 * \cdots * b_h$ all seen by p_i, and for some finite string τ, then $\texttt{chain}(\sigma) := b_0 * \cdots * b_h$. Otherwise, $\texttt{chain}(\sigma) := b_0$.
- $\texttt{reduct}(\sigma)$: If there exists a greatest $h \in \mathbb{N}$ such that $\sigma = H(b_0) * \cdots * H(b_h) * \tau$ for a chain of blocks $b_0 * \cdots * b_h$ all seen by p_i, and for some finite string τ, then $\texttt{reduct}(\sigma) := H(b_0) * \cdots * H(b_h)$. Otherwise, $\texttt{reduct}(\sigma) := H(b_0)$.
- $\texttt{last}(\sigma)$: If there exists a greatest $h \in \mathbb{N}$ such that $\sigma = H(b_0) * \cdots * H(b_h) * \tau$ for a chain $b_0 * \cdots * b_h$ all seen by p_i, and for some finite string τ, then $\texttt{last}(\sigma) := b_h$. Otherwise, $\texttt{last}(\sigma) := b_0$.
- H_B: If $B = b_0 * b_1 * \cdots * b_h$ is a chain, then $H_B := H(b_0) * H(b_1) * \dots H(b_h)$, and if not then H_B is the empty string $\emptyset$.

The pseudocode is described in Algorithm 2. For strings σ and τ, we write $\sigma \subseteq \tau$ to denote that σ is an initial segment of τ. For the sake of simplicity, the pseudocode considers a fixed value for α_2, but one could also incorporate approaches such as Error-Driven Snowflake$^+$ (described in Section 4.1).

6 CONSISTENCY ANALYSIS FOR SNOWMAN

We write $\texttt{pref}_i$ and $\texttt{final}_i$ to denote the values `pref` and `final` as locally defined for p_i. We say σ *becomes final* for p_i if there exists some round during which $\sigma \subseteq \texttt{final}_i$. We say σ *becomes final* if it becomes final for all correct processors. A block b becomes final if there exists some chain $B = b_0 * \cdots * b$ such that H_B becomes final.

The requirements. A probabilistic protocol for SMR is required to satisfy the following properties, except with small error probability:
Liveness: Infinitely many blocks become final.[5]
Consistency: Suppose $\sigma := \texttt{final}_i$ as defined at the beginning of round s and that $\sigma' := \texttt{final}_j$ as defined at the beginning of round s'. Then, whenever p_i and p_j are correct:

(i) If $i = j$ and $s' \geq s$ then $\sigma \subseteq \sigma'$.
(ii) Either σ extends σ', or σ' extends σ.

In this section, we show that Snowman satisfies consistency (except with small error probability) for appropriate choices of the protocol parameters, and so long as $f < n/5$. As in Section 4, for the sake of concreteness we give an analysis for $k = 80$, $\alpha_1 = 41$, $\alpha_2 = 72$, and $\beta = 12$, under the assumption that the population size $n \geq 500$. As before, we make the assumption that $n \geq 500$ only so as to be able to give as simple a proof as possible: a more fine-grained analysis for smaller n is the subject of future work. In Section 7 we will describe how to augment Snowman with a module guaranteeing liveness, which is formally analysed in Section 8.

[5]To ensure that transactions are not censored, it is natural also to require the stronger condition that infinitely blocks *produced by correct processors* become final. We initially consider the version of liveness stated above for the sake of simplicity, but describe how to deal with the stronger requirement in Section 8. In Section 8, we will also analyse the maximum time required to finalize new values.

Algorithm 2 Snowman: The instructions for processor p_i

```
Inputs
Δ, k, α_1, α_2, β ∈ ℕ
Local values
val(σ), initially undefined
count(σ), initially set to 0
rpref(j, s), initially undefined                          ▷ Records preferred chain of p_{j,s}
blocks, initially contains just b_0                       ▷ Automatically updated
pref, initially set to H(b_0)
final, initially set to H(b_0)

The instructions for round s, beginning at time 2Δs:
    Form sample sequence ⟨p_{1,s}, ... p_{k,s}⟩;          ▷ Sample with replacement
    For j ∈ [1, k], send s to p_{j,s};                     ▷ Ask p_{j,s} for preferred chain

    Wait Δ;

    For each j such that p_i has received s from p_j:
        Send (s, chain(pref)) to p_j;                      ▷ Report preferred chain to p_j

    Wait Δ;

    For each j ∈ [1, k]:
        If p_i has received a first message (s, B) from p_{j,s} s.t. B is a chain;
            Set rpref(j, s) := H_B;                        ▷ Record preferred chain of p_{j,s}
        Else set rpref(j, s) := H(b_0);

    Set pref := final, end := 0;        ▷ Begin iteration to determine pref for round s + 1
    While end == 0 do:
        Set E := {b ∈ blocks : b is a child of last(pref) and pref ⊆ reduct(pref) ∗ H(b)};
        If E is empty, set end := 1;
        Else:                                   ▷ Carry out the next instance of Snowflake+
            If val(pref) is undefined:
                Let b be the first block in E enumerated into block;
                Set val(pref) to be the (|pref| + 1)^th bit of reduct(pref) ∗ H(b);
            If |{j ∈ [1, k] : rpref(j, s) ⊇ pref ∗ 1 − val(pref)}| ≥ α_1:
                Set val(pref) := 1 − val(pref); For all σ ⊇ pref, set count(σ) := 0;
            If |{j ∈ [1, k] : rpref(j, s) ⊇ pref ∗ val(pref)}| < α_2:
                For all σ ⊇ pref, set count(σ) := 0;
            If |{j ∈ [1, k] : rpref(j, s) ⊇ pref ∗ val(pref)}| ≥ α_2:
                Set count(pref) := count(pref) + 1;
            If count(pref) ≥ β:
                Set final := pref ∗ val(pref);
            Set pref := pref ∗ val(pref);
```

The proof of consistency. It follows directly from the protocol instructions that (i) in the definition of consistency is satisfied. To see this, note that, initially, $\texttt{pref}_i = \texttt{final}_i = H(b_0)$. The values $\texttt{pref}_i$ and $\texttt{final}_i$ are not redefined during round s prior to line 27, when we set $\texttt{pref}_i := \texttt{final}_i$. If $\texttt{pref}_i$ is subsequently redefined during round s, then we redefine it to be an extension of its previous value. If $\texttt{final}_i$ is redefined during round s, then it is defined to be an extension of the present value of $\texttt{pref}_i$.

To argue that (ii) in the definition of consistency is satisfied, we again adopt the conventions described in Section 2 concerning the treatment of small error probabilities. Suppose inductively that σ becomes final for some correct processor and that, except with small error probability, no string incompatible with σ ever becomes final for any correct processor. This means we can condition on the event:

($\dagger_0$) No string incompatible with σ ever becomes final for any correct processor.

We must show that, if any string $\sigma * i$ (for $i \in \{0,1\}$) ever becomes final for a correct processor, then, except with small error probability, $\sigma * (1-i)$ does not become final for any correct processor. To achieve this, we argue almost exactly as in the analysis of Snowflake$^+$ in Section 4.

So, suppose that some correct processor p is the (potentially joint) first to set its local value $\texttt{final} \supset \sigma$, and, without loss of generality, suppose p sets $\texttt{final} \supseteq \sigma * 1$. To adjust the analysis of Section 4, we redefine the meaning of 'red' and 'blue'. We say correct processor p_i is 'red' in round s if $\texttt{pref}_i \supseteq \sigma * 1$ at the beginning of round s, and that p_i is 'blue' in round s otherwise – note that this includes the possibility that $\texttt{pref}_i$ is an initial segment of σ. We'll say a correct processor p_i 'samples x blue' in round s if $|\{j : 1 \leq j \leq k, \texttt{rpref}(j,s) \not\supseteq \sigma * 1\}| = x$ and that p_i 'samples x red' in round s if $|\{j : 1 \leq j \leq k, \texttt{rpref}(j,s) \supseteq \sigma * 1\}| = x$, where $\texttt{rpref}(j,s)$ is as locally defined for p_i at the end of round s. We'll say that p_i 'decides blue' if there is some round in which p_i sets $\texttt{final}_i$ to be σ' which is incompatible with $\sigma * 1$: Note that we do not include the possibility that $\sigma' \subseteq \sigma$ here and that, by ($\dagger_0$), if p_i is correct and decides blue then it sets $\texttt{final}_i \supseteq \sigma * 0$. We'll say p_i 'decides red' if it sets $\texttt{final}_i \supseteq \sigma * 1$.

As in Section 4, we start by considering what happens when the proportion of correct processors that are red reaches a certain threshold. In particular, let us consider what happens when at least 75% of the correct processors are red in a given round s such that no correct processor decides blue in any round $< s$. The binomial calculations then carry through just as in Section 4. A simple calculation for the binomial distribution shows that the probability a given correct processor is red in round $s+1$ is then at least 0.9555. To see this, note that if p_i samples at least 41 red in round s, then it must set $\texttt{pref}_i$ to be compatible with $\sigma * 1$ in line 27 during round s, and that the **while** loop (lines 28–43) will then set $\texttt{pref}_i$ to be an extension of $\sigma * 1$. Assuming a population of at least 500, of which at least 80% are correct, another simple calculation for the binomial distribution shows that the probability that it fails to be the case that more than 5/6 of the correct processors are red in round $s+1$ is upper bounded by 1.59×10^{-20}. For a population of at least 500, and by iterating this argument over rounds, we can therefore condition on the following event:

($\dagger_1$) If at least 75% of the correct processors are red in any round s such that no correct processor decides blue in any round $< s$, then, in all rounds s' with $s' > s$ such that no correct processor decides blue prior to s', more than 5/6 of the correct processors are red.

A simple calculation for the binomial distribution shows that if at least 75% of correct processors are red in a given round s, then the probability that a given correct processor p_i samples at least 72 blue in round s is upper bounded by 1.18×10^{-20}. We can therefore condition on the following event:

($\dagger_2$) If at least 75% of the correct processors are red in any round s, then no correct processor samples at least 72 blue in round s.

Since p_i cannot decide blue in a first round s unless it samples at least 72 blue in that round, and by combining ($\dagger_1$) and ($\dagger_2$), we may therefore condition on the event:

($\dagger_1'$) If at least 75% of the correct processors are red in any round s such that no correct processor decides blue in any round $< s$, then, in all rounds s' with $s' > s$, more than 5/6 of the correct processors are red.

Another simple calculation for the binomial distribution shows that, if *at most* 75% of correct processors are red in a given round s, then the probability a given correct processor samples 72 or more red in round s is upper bounded by 0.0131. So, if we consider 12 consecutive rounds in which at most 75% of the correct processors are red, the probability a given correct processor samples at least 72 red in each round is upper bounded by 10^{-22}. We can therefore condition on the following event:

($\dagger_3$) If a correct processor decides red in some first round $s + 11$, then, for at least one round $s' \in [s, s + 11]$, at least 75% of correct nodes are red in round s'.

Now recall that p is the (potentially joint) first processor to decide red or blue, and that p decides red. Suppose that p first decides red after sampling in round $s + 11$. According to ($\dagger_3$), at least one round $s' \in [s, s + 11]$ must satisfy the condition that at least 75% of correct processors are red in round s' and it must also hold that no correct processor decides blue prior to s'. Given our assumption ($\dagger_1'$), it follows that at least 5/6 of the correct processors must be red in all rounds $> s'$. From assumption ($\dagger_2$), it follows that no correct processor ever decides blue. This means $\sigma * (1 - i)$ does not become final for any correct processor, as required.

The accumulation of error probabilities. This can be bounded exactly as in Section 4. To see this, define σ_s to be the longest string such that at least 75% of correct processors have local `pref` values extending σ_s at the beginning of round s, and such that no correct processor finalizes any value incompatible with σ_s in any round $< s$. For the proof above to go through, we need only condition on the following holding for each s:

(1) More than 5/6 of the honest processors have local `pref` values extending σ_s by the end of round s.
(2) No correct processor samples at least 72 `pref` values that do not extend σ_s during round s.
(3) If any σ satisfies the condition that, for all $s' \in [s, s + 11]$, σ_s does not extend σ, then no correct processor samples at least 72 `pref` values extending σ in every round $s' \in [s, s + 11]$.

The numerical calculations can therefore be carried through with precisely the same bounds as in Section 4.

7 FROSTY

Recall that our next aim is to augment Snowman with a liveness module, allowing us to guarantee liveness in the case that $f < n/5$.

7.1 Overview of Frosty

In what follows, we assume that all messages are signed by the processor sending the message. We also suppose that $f < n/5$. Recall that the local variable `pref` is a processor's presently preferred chain and that `final` is its presently finalized chain.

The use of epochs. As outlined in Section 1, the basic idea is to run the Snowman protocol during standard operation, and to temporarily fall back to a standard 'quorum-based' protocol in the event that a substantial adversary attacks liveness for Snowman. We therefore consider instructions that are divided into *epochs*. In the first epoch (epoch 0), processors implement Snowman. In the event of a liveness attack, processors then enter epoch 1 and implement the quorum-based protocol to finalize the next block. Once this is achieved, they enter epoch 2 and revert to Snowman, and so on. Processors only enter each odd epoch and start implementing the quorum-based protocol if a liveness attack during the previous epoch forces them to do so.

Adding a decision condition. In even epochs, and when a processor sees sufficiently many consecutive rounds during which its local value `final` remains unchanged, it will send a message to others indicating that it wishes to proceed to the next epoch. Before any correct processor p_i enters the next epoch, however, it requires messages from at least 1/5 of all processors indicating that they wish to do the same. This is necessary to avoid the adversary being able to trigger a change of epoch at will, but produces a difficulty: some correct processors may wish to enter the next epoch, but the number who wish to do so may not be enough to trigger the epoch change. To avoid such a situation persisting for an extended duration, we introduce an extra decision condition. Processors now report their value `final` as well as their value `pref` when sampled. We consider an extra parameter α_3: for our analysis here, we suppose $\alpha_3 = 48$ (since $48 = \frac{3}{5} \cdot 80$). If p_i sees two consecutive samples in which at least α_3 processors report `final` values that all extend σ, then p_i will regard σ as final. For $k = 80$, $\alpha_3 = 48$ and if $f < n/5$, the probability that at least 3/5 of p_i's sample sequence in a given round are Byzantine is less than 10^{-14}, so the probability that this happens in two consecutive rounds is small. Except with small probability, the new decision rule therefore only causes p_i to finalize σ in the case that a correct processor has already finalized this value, meaning that it is safe for p_i to do the same. Using this new decision rule, we will be able to argue below that epoch changes are triggered in a timely fashion: either the epoch change is triggered soon after any correct p_i wishes to change epoch, or else sufficiently many correct processors do not wish to trigger the change that p_i is quickly able to finalize new values.

Epoch certificates. While in even epoch e, and for a parameter γ (chosen to taste), p_i will send the (signed) message $(\text{stuck}, e, \texttt{final})$ to all others when it sees γ consecutive rounds during which its local value `final` remains unchanged. This message indicates that p_i wishes to enter epoch $e + 1$ and is referred to as an 'epoch $e + 1$ message'. For any fixed σ, a set of messages of size at least $n/5$, each signed by a different processor and of the form $(\text{stuck}, e, \sigma)$, is called an *epoch certificate* (EC) for epoch $e + 1$.[6] When p_i sees an EC for epoch $e + 1$, it will send the EC to all others and enter epoch $e + 1$. This ensures that when any correct processor enters epoch $e + 1$, all others will do so within time Δ.

Ensuring consistency between epochs. We must ensure that the value finalized by the quorum-based protocol during an odd epoch $e+1$ extends all `final` values for correct processors. To achieve this, the rough idea is that we have processors send out their local `pref` values upon entering epoch $e + 1$, and then use these values to extract a chain that it is safe for the quorum based protocol to build on. Upon entering epoch $e + 1$, we therefore have p_i send out the message $(\text{start}, e + 1, \texttt{pref})$. This message is referred to as a *starting vote* for epoch $e + 1$ and, for any string σ, we say that the starting vote $(\text{start}, e + 1, \texttt{pref})$ extends σ if $\sigma \subseteq \texttt{pref}$. By a *starting certificate* (SC) for epoch $e + 1$ we mean a set of at least $2n/3$ starting votes for epoch $e + 1$, each signed by a different processor. If S is an SC for epoch $e + 1$, we set $\texttt{Pref}^*(S)$ to be the longest σ extended by more than half of the messages in S. The basic idea is that $\texttt{Pref}^*(S)$ must extend all `final` values for correct processors,

[6] To ensure ECs are strings of constant bounded length (independent of n), one could use standard 'threshold' cryptography techniques [7, 31], but we will not concern ourselves with such issues here.

and that consistency will therefore be maintained if we have the quorum-based protocol finalize a value extending this string.

To argue that this is indeed the case, recall the proof described in Section 6 (and recall that $f < n/5$). We argued there that, if any correct processor p_i finalizes σ in a given round, then (except with small error probability), more than 5/6 of the honest processors must have local `pref` values that extend σ by the end of that round, and that this will also be the case in all subsequent rounds. This might seem to ensure that $\texttt{Pref}^*(S)$ will extend σ: since $\frac{5}{6} \cdot \frac{4}{5} = \frac{2}{3}$, and since S contains at least $2n/3$ starting votes, it is tempting to infer that more than half the votes in S must extend σ. A complexity here, however, is that this reasoning only applies if all `Pref` values are reported *in the same round*. We can't (easily) ensure that all correct processors enter $e+1$ epoch in the same round, meaning that some correct processors may send their `Pref` values in one round, while others send them in the next round. To deal with this, we increase the β parameter from 12 to 14. This ensures (except with small error probability) that, when a correct processor p_i finalizes σ, more than 11/12 of correct processors have local `pref` values that extend σ by the end of the previous round, and that this is also true in all subsequent rounds. If s and $s+1$ are two consecutive rounds after p_i finalizes σ, and if we partition the correct processors arbitrarily so that some report their `pref` value in round s, while the rest do so in round $s+1$, then at least 5/6 of the correct processors must report values extending σ.

The choice of quorum-based protocol. While any of the standard quorum-based protocols could be implemented during odd epochs, for the sake of simplicity we give an exposition that implements a form of Tendermint, and we assume familiarity with that protocol in what follows. Let f^* be the greatest integer less than $n/3$. Recall that the instructions for Tendermint are divided into rounds (sometimes called 'views'). Within each round, there are two stages of voting, each of which is an opportunity for processors to vote on a block proposed by the *leader* of the round. The first stage of voting may establish a *stage 1 quorum certificate* (QC) for the proposed block, which is a set of stage 1 votes from $n - f^*$ distinct processors. In this event, the second stage may establish a *stage 2* QC for the block. The protocol also implements a *locking* mechanism. Processor p_i maintains a value $\texttt{Q}^+$. When they cast a stage 2 vote during round s, meaning that they have seen some Q which is a stage 1 QC for the proposal they are voting on, they set $Q^+ := Q$.

In our version of Tendermint, each leader will make a *proposal* P, and other processors will then vote on the proposal (so our 'proposals' play the role of blocks in Tendermint).

7.2 Further terminology

We consider the following new variables and other definitions (in addition to those used in previous sections).

f^*: The greatest integer less than $n/3$.

`e`: The epoch in which p_i is presently participating (initially 0).

`stuckcount`: Counts the number of consecutive rounds with `final` unchanged.

$\texttt{ready}(e)$: Indicates whether we have already initialized values for epoch e. Initially, $\texttt{ready}(e) = 0$. Processor p_i sets $\texttt{ready}(e) := 1$ upon entering epoch e after initializing values so that it is ready to start executing instructions for the epoch.

$\texttt{Init}(e)$: This process is run at the beginning of even epoch e, and performs the following: Set `pref` := `final`, `stuckcount` := 0, and for all σ set $\texttt{count}(\sigma) := 0$, $\texttt{primed}(\sigma) := 0$, and make $\texttt{val}(\sigma)$ undefined.

`M`: The set of all messages so far received by p_i.

$\texttt{lead}(s)$: The leader of round s while in an odd epoch. We set $\texttt{lead}(s) = p_j$, where $j = s \bmod n$.

$\texttt{primed}(\sigma)$: Used to help implement the new decision rule. This value is initially 0, and is set to 1 when p_i sees sufficiently many sampled `final` values extending σ. In the next round, p_i either finalizes σ (if the same holds again), or else resets $\texttt{primed}(\sigma)$ to 0.

Starting votes: A starting vote for epoch e is a message of the form $(\text{start}, e, \sigma)$ for some σ. For any string σ', we say that the starting vote $(\text{start}, e, \sigma)$ extends σ' if $\sigma' \subseteq \sigma$.

Starting certificates: A starting certificate for epoch e is set of at least $2n/3$ starting votes for epoch e, each signed by a different processor.

$\texttt{Pref}^*(S)$: If S is a starting certificate (SC) for epoch e, we set $\texttt{Pref}^*(S)$ to be the longest σ extended by more than half of the messages in S.

Votes. A vote V (for a proposal) is entirely specified by the following values:

$\texttt{P}(V)$: The proposal for which V is a vote.

$\texttt{st}(V)$: The stage of the vote (1 or 2).

The empty proposal. We call $\emptyset$ the *empty proposal*, and also let $\emptyset$ be a stage 1 QC for the empty proposal. We set $\texttt{r}(\emptyset) := 0$. The empty proposal is M*-valid* for any set of messages M.

Proposals. A proposal P other than the empty proposal is entirely specified by the following values:
$\texttt{r}(P)$: The round corresponding to the proposal.
$\texttt{e}(P)$: The epoch corresponding to the proposal.
$\texttt{par}(P)$: A proposal which is called the *parent* of P.[7]
$\texttt{QCprev}(P)$: A stage 1 QC for $\texttt{par}(P)$.
$\texttt{final}(P)$: The value that P attempts to finalize.
$\texttt{SC}(P)$: A starting certificate justifying the proposed value for finalization.

M**-valid proposals**. Let M be a set of messages. A proposal P other than the empty proposal is M-valid if it satisfies all of the following:

- $P \in M$.
- P has the empty proposal as an ancestor.
- $\texttt{par}(P)$ is M-valid.
- If $\texttt{par}(P)$ is not the empty proposal, then $\texttt{e}(P) = \texttt{e}(\texttt{par}(P))$.
- If $\texttt{par}(P)$ is not the empty proposal, then $\texttt{final}(P) = \texttt{final}(\texttt{par}(P))$.
- $\texttt{QCprev}(P)$ is a stage 1 QC for $\texttt{par}(P)$.
- $\texttt{SC}(P)$ is a starting certificate for epoch $\texttt{e}(P)$.
- $\texttt{final}(P)$ extends $\texttt{Pref}^*(\texttt{SC}(P))$.

An `M`**-valid proposal for round** s. Let `M`, `blocks` and `e` be as locally defined for p_i. While in round s, at time $3s\Delta + \Delta$, p_i will regard the proposal P as an `M`-valid proposal for round s if all of the following are satisfied:

- P is `M`-valid.
- $\texttt{r}(P) = s$ and P is signed by $\texttt{lead}(s)$.
- $\texttt{e}(P) = \texttt{e}$.
- There exists a chain $B = b_0 * \cdots * b_h$ such that $\texttt{final}(P) = H_B$ and, for $j \in [1, h]$, $b_j \in \texttt{blocks}$.

[7]We adopt similar terminology for proposals and blocks: If P' is the parent of P, then P is referred to as a *child* of P. In this case, the ancestors of P are P and any ancestors of P'. The descendants of any proposal P are P and any descendants of its children.

M-**confirmed proposals**. For any set of messages M, a proposal P is M-confirmed if a descendant P' of P (possibly P itself) is M-valid and M contains a stage 2 QC for P'.

Epoch certificates. For any fixed σ, a set of messages of size at least $n/5$, each signed by a different processor and of the form $(\text{stuck}, e, \sigma)$, is called an *epoch certificate* (EC) for epoch $e+1$.

Quorum certificates. If P is any proposal other than the empty proposal, then, by a stage x QC for P, we mean a set of votes Q of size $n - f^*$, each signed by a different processor, and such that $\mathtt{P}(V) = P$ and $\mathtt{st}(V) = x$ for each $V \in Q$. If Q is a (stage 1 or 2) QC for P, then we define $\mathtt{r}(Q) := \mathtt{r}(P)$.

The procedure `MakeProposal`. If $p_i = \mathtt{lead}(s)$, then this procedure is used by p_i while in odd epochs to send an appropriate proposal to all:

- If p_i has not seen an SC for epoch e, then do not send any proposal. Otherwise, let S be such an SC and proceed as follows.
- Let s' be the greatest such that $\mathtt{M}$ contains an $\mathtt{M}$-valid proposal P' with $\mathtt{r}(P') = s'$, $\mathtt{e}(P') = \mathtt{e}$ if $s' > 0$, and such that $\mathtt{M}$ also contains Q which is a stage 1 QC for P'.
- Set $\mathtt{r}(P) := s$, $\mathtt{e}(P) := \mathtt{e}$, $\mathtt{par}(P) := P'$, $\mathtt{QCprev}(P) := Q$.
- If $s' = 0$, i.e. if P' is the empty proposal, then proceed as follows. Set $\mathtt{SC}(P) := S$. Let $B = b_0 * \cdots * b_h$ be a chain such that H_B extends $\mathtt{Pref}^*(S)$ and, for $j \in [1, h]$, $b_j \in \mathtt{blocks}$ (if there exists no such B then do not send a proposal). Set $\mathtt{final}(P) := H_B$.
- If $s' > 0$, then set $\mathtt{SC}(P) := \mathtt{SC}(P')$ and $\mathtt{final}(P) := \mathtt{final}(P')$.
- Send P to all.

Conventions regarding the gossiping of blocks, proposals and QCs while in an odd epoch. While in odd epochs, we suppose that correct processors automatically gossip received blocks, proposals and QCs for proposals. This means that if p_i is correct and sees a QC (for example) at time t, then all correct processors see that QC by time $t + \Delta$. When p_i sends a message to all, it is also convenient to assume p_i regards that message as received (by p_i) at the next timeslot.

For ease of reference, inputs and local variables are listed in the table below.

Frosty: The inputs and local values for processor p_i
Inputs
$\Delta, k, \alpha_1, \alpha_2, \alpha_3, \beta, \gamma \in \mathbb{N}$
Local values
$\mathtt{val}(\sigma)$, initially undefined.
$\mathtt{count}(\sigma)$, initially set to 0
$\mathtt{primed}(\sigma)$, initially set to 0
`stuckcount`, initially set to 0
$\mathtt{rpref}(j, s)$, initially undefined
$\mathtt{final}(j, s)$, initially undefined
`blocks`, initially contains just b_0
`pref`, initially set to $H(b_0)$
`final`, initially set to $H(b_0)$
`e`, initially set to 0
$\mathtt{ready}(e)$, initially set to 0 for all $e \in \mathbb{N}_{\geq 0}$
$\mathtt{Q}^+$, initially set to $\emptyset$
`M`, initially contains just b_0
`P`, initially undefined

Algorithm 3 Frosty: The instructions for processor p_i **while** $\mathtt{e}$ is even

At every t **if** $\mathtt{ready(e)} == 0$ **then** $\mathtt{Init}(e)$, set $\mathtt{ready}(e) := 1$; ▷ Initialise values for epoch $\mathtt{e}$

At $t = 3\Delta s$ **if** $\mathtt{ready(e)} == 1$: ▷ For any $s \in \mathbb{N}_{\geq 1}$
  Form sample sequence $\langle p_{1,s}, \dots p_{k,s} \rangle$; ▷ Sample with replacement
  For $j \in [1, k]$, send $(s, \mathtt{e})$ to $p_{j,s}$; ▷ Ask $p_{j,s}$ for preferred chain
  If $\mathtt{M}$ contains an EC for epoch $\mathtt{e} + 1$: send the EC to all, set $\mathtt{e} := \mathtt{e} + 1$; ▷ Enter next epoch

At $t = 3\Delta s + \Delta$ **if** $\mathtt{ready(e)} == 1$:
  For each j such that p_i has received $(s, \mathtt{e})$ from p_j:
    Send $(s, \mathtt{e}, \mathtt{chain(pref)}, \mathtt{chain(final)})$ to p_j; ▷ Report present values to p_j
  If $\mathtt{M}$ contains an EC for epoch $\mathtt{e} + 1$: send the EC to all, set $\mathtt{e} := \mathtt{e} + 1$; ▷ Enter next epoch

At $t = 3\Delta s + 2\Delta$ **if** $\mathtt{ready(e)} == 1$:
  Form sample sequence $\langle p_{1,s}, \dots p_{k,s} \rangle$ if not already formed.
  For each $j \in [1, k]$:
    If p_i has received a first message $(s, \mathtt{e}, B_1, B_2)$ from $p_{j,s}$ s.t. B_1, B_2 are chains;
      Set $\mathtt{rpref}(j, s) := H_{B_1}$, $\mathtt{final}(j, s) := H_{B_2}$; ▷ Record values from $p_{j,s}$
    Else set $\mathtt{rpref}(j, s) := H(b_0)$, $\mathtt{final}(j, s) := H(b_0)$;
  If $\mathtt{M}$ contains an EC for epoch $\mathtt{e} + 1$: send the EC to all, set $\mathtt{e} := \mathtt{e} + 1$; ▷ Enter next epoch

At $t = 3\Delta s + 2\Delta$ **if** $\mathtt{ready(e)} == 1$: ▷ Execute instructions above first and re-check $\mathtt{ready(e)}$
  Set $E^* := \{b \in \mathtt{block} : \ b$ is a child of $\mathtt{last(final)}\}$;
  Set $\mathtt{pref} := \mathtt{final}$, $\mathtt{end} := 0$. If E^* is non-empty: $\mathtt{stuckcount}++$;
  While $\mathtt{end} == 0$ **do**: ▷ Iteration to determine $\mathtt{pref}$ for round $s + 1$
    Set $E := \{b \in \mathtt{block} : \ b$ is a child of $\mathtt{last(pref)}$ and $\mathtt{pref} \subseteq \mathtt{reduct(pref)} * H(b)\}$;
    If E is empty, set $\mathtt{end} := 1$;
    Else: ▷ Carry out the next instance of Snowflake$^+$
      If $\mathtt{val(pref)}$ is undefined:
        Let b be the first block in E enumerated into $\mathtt{block}$;
        Set $\mathtt{val(pref)}$ to be the $(|\mathtt{pref}| + 1)^{\text{th}}$ bit of $\mathtt{reduct(pref)} * H(b)$;
      If $|\{j \in [1, k] : \mathtt{rpref}(j, s) \supseteq \mathtt{pref} * 1 - \mathtt{val(pref)}\}| \geq \alpha_1$:
        Set $\mathtt{val(pref)} := 1 - \mathtt{val(pref)}$; For all $\sigma \supseteq \mathtt{pref}$, set $\mathtt{count}(\sigma) := 0$;
      If $|\{j \in [1, k] : \mathtt{rpref}(j, s) \supseteq \mathtt{pref} * \mathtt{val(pref)}\}| < \alpha_2$:
        For all $\sigma \supseteq \mathtt{pref}$, set $\mathtt{count}(\sigma) := 0$;
      If $|\{j \in [1, k] : \mathtt{rpref}(j, s) \supseteq \mathtt{pref} * \mathtt{val(pref)}\}| \geq \alpha_2$: $\mathtt{count(pref)}++$;
      If $\mathtt{count(pref)} \geq \beta$:
        Set $\mathtt{final} := \mathtt{pref} * \mathtt{val(pref)}$, $\mathtt{stuckcount} := 0$; ▷ Finalize and reset $\mathtt{stuckcount}$
      If $\mathtt{count(pref)} < \beta$ **then** for $x \in \{0, 1\}$ **do**: ▷ New decision rule
        If $|\{j \in [1, k] : \mathtt{final}(j, s) \supseteq \mathtt{pref} * x\}| \geq \alpha_3$:
          If $\mathtt{primed}(\mathtt{pref} * x) == 1$; ▷ previous round primed $\mathtt{pref} * x$ for finalization
            Set $\mathtt{final} := \mathtt{pref} * x$, $\mathtt{stuckcount} := 0$; ▷ Finalize and reset $\mathtt{stuckcount}$
          Else set $\mathtt{primed}(\mathtt{pref} * x) := 1$; ▷ Prime $\mathtt{pref} * x$ for finalization
        Else set $\mathtt{primed}(\mathtt{pref} * x) := 0$;
      Set $\mathtt{pref} := \mathtt{pref} * \mathtt{val(pref)}$;
  If $\mathtt{stuckcount} \geq \gamma$ **then** send $(\text{stuck}, \mathtt{e}, \mathtt{final})$ to all; ▷ After completing **while** loop

Algorithm 4 Frosty: The instructions for processor p_i **while** $\mathtt{e}$ is odd

At every t **if** $\mathtt{ready(e)} == 0$ **then**:
    Send (start, $\mathtt{e}, \mathtt{pref}$) to all; ▷ Send starting vote
    Set $\mathtt{Q}^+ := \emptyset$; ▷ Initialize $\mathtt{Q}^+$
    Set $\mathtt{ready}(e) := 1$;

At every t, **if** there exists proposal $P \in \mathtt{M}$ with $\mathtt{e}(P) == \mathtt{e}$ which is $\mathtt{M}$-confirmed **then**:
    Set $\mathtt{final} := \mathtt{final}(P)$; ▷ Finalize next block
    Set $\mathtt{e} := \mathtt{e} + 1$; ▷ Enter next epoch. Others will do so within Δ (due to gossiping)

At $t = 3\Delta s$ **if** $\mathtt{lead}(s) == i$: ▷ For any $s \in \mathbb{N}_{\geq 1}$
    $\mathtt{MakeProposal}$;

At $t = 3\Delta s + \Delta$:
    Set $\mathtt{P}$ to be undefined.
    If p_i has received a first $\mathtt{M}$-valid proposal P for round s **and** $\mathtt{r}(\mathtt{QCprev}(P)) \geq \mathtt{r}(\mathtt{Q}^+)$:
        Set $\mathtt{P} := P$;
        Send vote V to all, with $\mathtt{P}(V) := \mathtt{P}$, $\mathtt{st}(V) := 1$; ▷ Send stage 1 vote

At $t = 3\Delta s + 2\Delta$:
    If $\mathtt{P}$ is defined and p_i has received a first Q which is a stage 1 QC for $\mathtt{P}$:
        Set $\mathtt{Q}^+ := Q$; ▷ Set lock
        Send vote V to all, with $\mathtt{P}(V) := \mathtt{P}$, $\mathtt{st}(V) := 2$; ▷ Send stage 2 vote

8 FROSTY: CONSISTENCY AND LIVENESS ANALYSIS

We give an analysis for the case that $k = 80$, $\alpha_1 = 41$, $\alpha_2 = 72$, $\alpha_3 = 48$, $\beta = 14$, $\gamma \geq 300$, and under the assumption that $n \geq 500$ and $f < n/5$. As before, we make the assumption that $n \geq 500$ only so as to be able to give as simple a proof as possible: a more fine-grained analysis for smaller n is the subject of future work.

8.1 The proof of consistency

Section 6 already established that Snowman satisfies consistency (except with small error probability). To establish consistency for Frosty, we must show that, if an odd epoch e finalizes any value, then it finalizes a single value extending any values finalized by correct processors during previous epochs. Before considering the instructions during odd epochs, however, there are three new complexities with respect to the instructions during an even epoch e, which we must check cannot lead to a consistency violation during the same epoch:

(i) The new decision rule.
(ii) Players may not enter epoch $e + 1$ entirely simultaneously. Could this impact the samples received by p_i in some round and cause a consistency violation?
(iii) Players may not enter epoch e entirely simultaneously. This possibility did not arise in Section 6, so could this be the cause of a consistency violation?

Dealing with (i). Suppose p_i is correct. A simple calculation for the binomial distribution shows that the probability that at least 3/5 of p_i's sample sequence in a given round are Byzantine is less than 10^{-14}. The probability that this happens in two given consecutive rounds is therefore less than

10^{-28}. We may therefore condition on the following event (letting $\texttt{final}(j, s)$ and $\texttt{final}(j, s+1)$ be as locally defined for p_i at the end of rounds s and $s+1$ respectively):

($\diamond_0$): If there exists s and σ such that $|\{j \in [1,k] : \texttt{final}(j,s) \supseteq \sigma\}| \geq \alpha_3$ and $|\{j \in [1,k] : \texttt{final}(j,s+1) \supseteq \sigma\}| \geq \alpha_3$, then some correct processor has already finalized a value extending σ by the end of round s.

Conditioned on ($\diamond_0$), it is not possible for the new decision rule to cause a first consistency violation.

Dealing with (ii). If any correct processor enters epoch $e+1$ at t, then they send an EC for epoch $e+1$ to all. All correct processors therefore enter epoch $e+1$ by $t+\Delta$. In particular, this means that if correct p_i is in epoch e at time $3s\Delta + 2\Delta$ (when considering values reported during round s), no correct processor will have entered epoch $e+1$ prior to reporting their values during round s (lines 8–11 of the pseudocode). The distribution on reported values is thus unaffected by the fact that some correct processors may have already entered epoch $e+1$ at that point.

Dealing with (iii). If $e > 0$ is even and some correct processor enters epoch e at t because there exists $P \in \texttt{M}$ with $\texttt{e}(P) = e-1$ which is $\texttt{M}$-confirmed, then (due to the gossiping of blocks and QCs) all correct processors will enter epoch e by $t+\Delta$. The arguments of Section 6 are unaffected if we allow that some correct processors only begin executing instructions midway through the first round of an epoch (this just means that some correct processors might not report values or ask for values in the first round).

Let $\texttt{M}$ be as locally defined for correct p_i and say that a proposal P *becomes confirmed for* p_i if there is some timeslot at which P is $\texttt{M}$-confirmed. Suppose e is odd and that no consistency violation occurs prior to the first point at which any correct processor enters epoch e. To complete the proof of consistency, it suffices to establish the two following claims:

Claim 1. If the proposal P with $\texttt{e}(P) = e$ becomes confirmed for correct p_i, then $\texttt{final}(P)$ extends any values finalized by correct processors during previous epochs.

Claim 2. If P and P' are proposals with $\texttt{e}(P) = \texttt{e}(P') = e$, and if P becomes confirmed for correct p_i and P' becomes confirmed for correct p_j, then $\texttt{final}(P) = \texttt{final}(P')$.

Establishing Claim 1. If the proposal P becomes confirmed for correct p_i, then some correct processors must produce votes for the proposal, which implies that:

- $\texttt{SC}(P)$ is a starting certificate for epoch e.
- $\texttt{final}(P)$ extends $\texttt{Pref}^*(\texttt{SC}(P))$.

We argued in Section 6 that, when $\beta = 12$, if some correct processor is the first to finalize the value σ and does so in some round $s+11$, say, then at least one round $s' \in [s, s+11]$ must satisfy the condition that at least 75% of correct processors have local $\texttt{pref}$ values extending σ in round s' and that at least 5/6 of the correct processors must have local $\texttt{pref}$ values extending σ (by the end of round s' and) in all rounds $> s'$. Now consider Frosty for the case that $\beta = 14$. Suppose that some correct processor is the first to finalize the value σ while in epoch $e-1$ and does so in some round $s+13$ (the case that no processor finalizes any new values while in epoch $e-1$ is similar). Then, by the same reasoning as in Section 6, at least one round $s' \in [s, s+11]$ must satisfy the condition that at least 5/6 of correct processors have local $\texttt{pref}$ values extending σ at the end of round s'. Let s'' be the greatest round such that some correct processor completes round s'' before entering epoch e. A simple calculation for the binomial then shows that, except with small error probability, at the end of each round in $[s+12, s'']$ (and even if some correct processors have already moved to epoch e during round s''), at least 11/12 of the correct processors must have local $\texttt{pref}$ values extending

σ. Note that the local `pref` values reported by correct processors in the starting certificate $\mathtt{SC}(P)$ are either those defined at the end of round s'' or round $s''-1$. We conclude that at least 5/6 of correct processors must send starting votes of the form $(\text{start}, e, \mathtt{pref})$ such that `pref` extends σ. Since $\mathtt{SC}(P)$ contains at least $2n/3$ starting votes, more than half the votes in $\mathtt{SC}(P)$ must extend σ, so that $\mathtt{final}(P)$ extends σ, as required.

Establishing Claim 2. Towards a contradiction, suppose there exists some least s and some least $s' \geq s$ such that:

- Some proposal P with $\mathtt{e}(P) = e$ and $\mathtt{r}(P) = s$ receives stage 1 and 2 QCs, Q_1 and Q_2 respectively;
- Some proposal P' with $\mathtt{e}(P') = e$ and $\mathtt{r}(P) = s'$ receives a stage 1 QC, Q'_1;
- $\mathtt{final}(P) \neq \mathtt{final}(P')$.

Suppose first that $s = s'$. Then, since each QC contains votes from at least $n - f^*$ distinct processors, some correct processor must produce votes in both Q_1 and Q'_1. This gives an immediate contradiction, because correct processors do not produce more than one stage 1 vote in any single round.

So, suppose that $s' > s$. In this case, some correct processor p_i must produce votes in both Q_2 and Q'_1. Since $\mathtt{final}(P) \neq \mathtt{final}(P')$, our choice of s and s' implies that $\mathtt{r}(\mathtt{QCprev}(P')) < s$. We reach a contradiction because p_i sets its lock $\mathtt{Q}^+$ so that $\mathtt{r}(\mathtt{Q}^+) = s$ while in round s, and so would not vote for P' in round s' (line 15 of the pseudocode).

The accumulation of small error probabilities. The analysis is the same as in Section 6, except that we must now account for two new assumptions on which we have conditioned in the argument above. Previously, we assumed that if at least 75% of the correct processors have local `pref` values extending σ at the beginning of round s, then at least 5/6 of the correct processors will have `pref` values extending σ by the end of round s. Now, we require the additional assumption that if 5/6 of the correct processors have local `pref` values extending σ at the beginning of round s, then at least 11/12 of the correct processors will have `pref` values extending σ by the end of round s. For a given round s, a simple calculation for the binomial distribution shows that this holds, except with probability at most 2×10^{-47}. If there are five rounds per second then, over a period of 1000 years, this means that less than 1.6×10^{11} rounds are executed. Applying the union bound, we conclude that this adds less than 4×10^{-36} to the cumulative error probability.

We must also account for the new decision condition. As noted previously, a simple calculation for the binomial distribution shows that if p_i is correct then the probability that at least 3/5 of p_i's sample sequence in a given round are Byzantine is less than 10^{-14}. The probability that this happens in two given consecutive rounds is therefore less than 10^{-28}. If at most 10,000 processors execute at most 5 rounds per second for 1000 years, this therefore adds less than 2×10^{-13} to the cumulative error probability.

Overall, the same error bound of 3×10^{-5} that was established in Section 4 can be seen to hold here.

8.2 The proof of liveness

Throughout this section, we assume that the value E^* (specified in line 22 of the pseudocode for even epochs) is never empty for correct p_i, i.e. there are always new blocks to finalize.

Defining $\mathtt{final}_t$. At any timeslot t, let $\mathtt{final}_t$ be the shortest amongst all local values `final` for correct processors (by Section 8.1 this value is uniquely defined, except with small error probability).

The proof of liveness breaks into two parts:

Claim 3. Suppose that all correct processors are in even epoch e at t. Then, except with small error probability, either $\texttt{final}_{t+6\Delta\gamma}$ properly extends $\texttt{final}_t$ or else all correct processors enter epoch $e+1$ by time $t+6\Delta\gamma$.

Claim 4. If all correct processors enter the odd epoch $e+1$, this epoch finalizes a new value.

In Claim 4, the number of rounds during epoch $e+1$ required to finalize a new value is bounded by the maximum number of consecutive faulty leaders. Since we make the simple choice of using deterministic leader selection during odd epochs, this means that the number of required rounds is $O(f)$, but one could ensure the number of required rounds is $O(1)$ and maintain a small chance of liveness failure by using random leader selection.

Establishing Claim 3. Given the conditions in the statement of the claim, let $E^\diamond$ be the set of correct processors that have local $\texttt{final}$ values properly extending $\texttt{final}_t$ at time $t_1 := t+3\Delta\gamma$. If $|E^\diamond| \leq 3n/5$, then at least $n/5$ correct processors send epoch $e+1$ messages $(\text{stuck}, e, \texttt{final}_t)$ by time t_1, and all correct processors enter epoch $e+1$ by time $t_1+\Delta$. So, suppose $|E^\diamond| > 3n/5$ and that it is not the case all correct processors enter epoch $e+1$ by time $t+6\Delta\gamma$. Let $x \in \{0,1\}$ be such that some correct processor finalizes $\texttt{final}_t * x$, and (by consistency) condition on there existing a unique such x. Consider the instructions as locally defined for correct p_i when executing any round s of epoch e that starts subsequent to t_1. A simple calculation for the binomial shows that the probability $|\{j \in [1,k] : \texttt{final}(j,s) \supseteq \texttt{final}_t * x\}| \geq 48$ is at least 0.548. The probability that this holds in both of any two such consecutive rounds s and $s+1$ is therefore at least 0.3. Since we suppose $\gamma \geq 300$, the probability that p_i fails to finalize a value extending $\texttt{final}_t$ by time $t+6\Delta\gamma$ is therefore at most $0.71^{150} < 10^{-22}$.

Establishing Claim 4. Towards a contradiction, suppose that all correct processors enter odd epoch $e+1$, but that the epoch never finalizes a new value. Let $t = 3\Delta s$ be such that $\texttt{lead}(s)$ is correct and all correct processors are in epoch $e+1$ at t, with their local value $\texttt{ready}(e+1)$ equal to 1. Let s' be the greatest such that any correct processor has a local value $\texttt{Q}^+$ at t with $\texttt{r}(\texttt{Q}^+) = s'$. Note that either $s' = 0$, or else any correct processor that has set is local value $\texttt{Q}^+$ so that $\texttt{r}(\texttt{Q}^+) = s'$, did so at a timeslot $\leq t-\Delta$. According to our conventions regarding the gossiping of QCs, this means that $\texttt{lead}(s)$ will receive a QC, Q say, with $\texttt{r}(Q) \geq s'$ by t. The correct processor $\texttt{lead}(s)$ will then send out a proposal P during round s that will be regarded as an $\texttt{M}$-valid proposal for round s by all correct processors. If $\texttt{Q}^+$ is as locally defined for any correct processor at $t+\Delta$, P will also satisfy the condition that $\texttt{r}(\texttt{QCprev}(P)) \geq \texttt{r}(\texttt{Q}^+)$. All correct processors will therefore send stage 1 and 2 votes for P, and P will be confirmed for all correct processors.

A comment on the finalization of blocks produced by correct leaders. For the sake of simplicity, we have structured the instructions for odd epochs so as to ensure the finalization of one more block, rather than so as to ensure the finalization of at least one more block produced by a correct leader. Of course, one could achieve the latter result simply by running odd epochs until at least $f+1$ distinct leaders have produced finalized blocks.

9 RELATED WORK

The Snow family of consensus protocol was introduced in [28]. Subsequent to this, Amores-Sesar, Cachin and Tedeschi [3] gave a complete description of the Avalanche protocol[8] and formally established security properties for that protocol, given an $O(\sqrt{n})$ adversary and assuming that the

[8]The Avalanche protocol is a DAG-based variant of Snowman that does not aim to produce a total ordering on transactions, and was only described at a high level in [28]. It is not used in the present instantiation of the Avalanche blockchain.

Snowball protocol (a variant of Snowflake$^+$) solves probabilistic Byzantine Agreement for such adversaries. The authors also described (and provided a solution for) a liveness attack. As noted in [3], the original implementation of the Avalanche protocol used by the Avalanche blockchain (before replacing Avalanche with a version of Snowman that totally orders transactions) had already introduced modifications avoiding the possibility of such attacks.

In [2], Amores-Sear, Cachin and Schneider consider the Slush protocol and show that coming close to a consensus already requires a minimum of $\Omega\left(\frac{\log n}{\log k}\right)$ rounds, even in the absence of adversarial influence. They show that Slush reaches a stable consensus in $O(\log n)$ rounds, and that this holds even when the adversary can influence up to $O(\sqrt{n})$ processors. They also show that the $\Omega\left(\frac{\log n}{\log k}\right)$ lower bound holds for Snowflake and Snowball.

There is a vast literature that considers a closely related family of models, from the *Ising model* [9] as studied in statistical mechanics, to *voter models* [18] as studied in applied probability and other fields, to the *Schelling model of segregation* [29] as studied by economists (and more recently by computer scientists [4, 8] and physicists [22–24]). Within this family of models there are many variants, but a standard approach is to consider a process that proceeds in rounds. In each round, each participant samples a small number of other participants to learn their present state, and then potentially updates their own state according to given rule. A fundamental difference with our analysis here is that, with two exceptions (mentioned below), such models do not incorporate the possibility of Byzantine action. Examples of such research aimed specifically at the task of reaching consensus include [6, 11–13, 15, 17] (see [5] for an overview). Amongst these papers, we are only aware of [6] and [13] considering Byzantine action, and those two papers deal only with an $O(\sqrt{n})$ adversary.

FPC-BI [26, 27] is a protocol which is closely related to the Snow family of consensus protocols, but which takes a different approach to the liveness issue (for adversaries which are larger than $O(\sqrt{n})$) than that described here. The basic idea behind their approach is to use a common random coin to dynamically and unpredictably set threshold parameters (akin to α_1 and α_2 here) for each round, making it much more difficult for an adversary to keep the honest population split on their preferred values. Since the use of a common random coin involves practical trade-offs, their approach and ours may be seen as complementary.

10 FINAL COMMENTS

In this paper, we have considered the case that the adversary controls at most $f < n/5$ processors. We described the protocol Snowflake$^+$ and showed that it satisfies validity and agreement, except with small error probability. We showed how Snowflake$^+$ can be adapted to give an SMR protocol, Snowman, which satisfies consistency, except with small error probability. We then augmented Snowman with a liveness module, to form the protocol Frosty, which we proved satisfies liveness and consistencty except with small error probability. We note that Avalanche presently implements Snowflake, rather than Snowflake$^+$, and uses different parameters than those used in the proofs here. Snowflake$^+$ was implemented a few months prior to the writing of this paper, but is not yet activated. Error-driven Snowflake$^+$ is planned for implementation in the coming months. The community may consider adopting the parameters proposed in this paper because they provide a good tradeoff between consistency and latency.

In future work, we aim to expand the analysis here as follows:

(i) The bounds $f < n/5$ and $n \geq 500$ were used only so as to be able to give as simple a proof as possible in Section 4. In subsequent papers, we intend to carry out a more fine-grained analysis for smaller n and larger f.
(ii) The analysis here was simplified by the assumption that processors execute instructions in synchronous rounds. In a follow-up work, we aim to show how the methods described here can be adapted to give formal proofs of consistency and liveness for a *responsive* form of the protocol, allowing each processor to proceed individually through rounds as fast as network delays allow. A promising approach in this direction is to add round numbers to queries, so that correct processors report their values for specific requested rounds rather than their present preferred values, thus enabling a responsive protocol to simulate the form of 'synchronous progression' protocol considered here.
(iii) While the liveness module described here achieves (probabilistic) liveness when $f < n/5$, we aim to explore ways in which *slashing* can be implemented for liveness attacks. This may be possible if one can show that liveness attacks require the adversary either to give provably false information to others, or else execute sampling that is provably biased.

11 ACKNOWLEDGEMENTS

The authors would like to thank Christian Cachin, Philipp Schneider, and Ignacio Amores Sesar for a number of very useful conversations.

REFERENCES

[1] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326, 2019.
[2] Ignacio Amores-Sesar, Christian Cachin, and Philipp Schneider. An analysis of avalanche consensus. *arXiv preprint arXiv:2401.02811*, 2024.
[3] Ignacio Amores-Sesar, Christian Cachin, and Enrico Tedeschi. When is spring coming? a security analysis of avalanche consensus. *arXiv preprint arXiv:2210.03423*, 2022.
[4] George Barmpalias, Richard Elwes, and Andy Lewis-Pye. Digital morphogenesis via schelling segregation. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 156–165. IEEE, 2014.
[5] Luca Becchetti, Andrea Clementi, and Emanuele Natale. Consensus dynamics: An overview. *ACM SIGACT News*, 51(1):58–104, 2020.
[6] Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Luca Trevisan. Stabilizing consensus with many opinions. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 620–635. SIAM, 2016.
[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.
[8] Christina Brandt, Nicole Immorlica, Gautam Kamath, and Robert Kleinberg. An analysis of one-dimensional schelling segregation. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 789–804, 2012.
[9] Stephen G Brush. History of the lenz-ising model. *Reviews of modern physics*, 39(4):883, 1967.
[10] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.
[11] Colin Cooper, Robert Elsässer, and Tomasz Radzik. The power of two choices in distributed voting. In *International Colloquium on Automata, Languages, and Programming*, pages 435–446. Springer, 2014.
[12] Emilio Cruciani, Hlafo Alfie Mimun, Matteo Quattropani, and Sara Rizzo. Phase transitions of the k-majority dynamics in a biased communication model. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, pages 146–155, 2021.
[13] Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing consensus with the power of two choices. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 149–158, 2011.
[14] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.
[15] Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Frederik Mallmann-Trenn, and Horst Trinker. Brief announcement: rapid asynchronous plurality consensus. In *Proceedings of the ACM symposium on principles of distributed computing*,

pages 363–365, 2017.

[16] Juan A Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. 2018.

[17] Mohsen Ghaffari and Johannes Lengler. Nearly-tight analysis for 2-choice and 3-majority consensus dynamics. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 305–313, 2018.

[18] Richard A Holley and Thomas M Liggett. Ergodic theorems for weakly interacting infinite systems and the voter model. *The annals of probability*, pages 643–663, 1975.

[19] Valerie King and Jared Saia. Breaking the o (n 2) bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint arXiv:2304.14701*, 2023.

[22] Hamed Omidvar and Massimo Franceschetti. Self-organized segregation on the grid. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 401–410, 2017.

[23] Hamed Omidvar and Massimo Franceschetti. Improved intolerance intervals and size bounds for a schelling-type spin system. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(7):073302, 2021.

[24] Diego Ortega, Javier Rodríguez-Laguna, and Elka Korutcheva. A schelling model with a variable threshold in a closed city segregation model. analysis of the universality classes. *Physica A: Statistical Mechanics and its Applications*, 574:126010, 2021.

[25] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. *Cryptology ePrint Archive*, 2016.

[26] Serguei Popov and William J Buchanan. Fpc-bi: Fast probabilistic consensus within byzantine infrastructures. *Journal of Parallel and Distributed Computing*, 147:77–86, 2021.

[27] Serguei Popov and Sebastian Müller. Voting-based probabilistic consensuses and their applications in distributed ledgers. *Annals of Telecommunications*, pages 1–23, 2022.

[28] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. *arXiv preprint arXiv:1906.08936*, 2019.

[29] Thomas C Schelling. Models of segregation. *The American economic review*, 59(2):488–493, 1969.

[30] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[31] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pages 207–220. Springer, 2000.