Simulating Population Protocols in Sub-Constant Time per Interaction

Petra Berenbrink
Universität Hamburg, Germany
petra.berenbrink@uni-hamburg.de

David Hammer
University of Southern Denmark, Odense, Denmark
Goethe University Frankfurt, Germany
hammer@imada.sdu.dk

Dominik Kaaser
Universität Hamburg, Germany
dominik.kaaser@uni-hamburg.de

Ulrich Meyer
Goethe University Frankfurt, Germany
umeyer@ae.cs.uni-frankfurt.de

Manuel Penschuck
Goethe University Frankfurt, Germany
mpenschuck@ae.cs.uni-frankfurt.de

Hung Tran
Goethe University Frankfurt, Germany
hung@ae.cs.uni-frankfurt.de

Abstract

We consider the efficient simulation of population protocols. In the population model, we are given a system of n agents modeled as identical finite-state machines. In each step, two agents are selected uniformly at random to interact by updating their states according to a common transition function. We empirically and analytically analyze two classes of simulators for this model. First, we consider sequential simulators executing one interaction after the other. Key to the performance of these simulators is the data structure storing the agents' states. For our analysis, we consider plain arrays, binary search trees, and a novel Dynamic Alias Table data structure. Secondly, we consider batch processing to efficiently update the states of multiple independent agents in one step. For many protocols considered in literature, our simulator requires amortized sub-constant time per interaction and is fast in practice: given a fixed time budget, the implementation of our batched simulator is able to simulate population protocols several orders of magnitude larger compared to the sequential competitors, and can carry out 2^{50} interactions among the same number of agents in less than 400 s.

2012 ACM Subject Classification Computing methodologies → Agent / discrete models

Keywords and phrases Population Protocols, Simulation, Random Sampling, Dynamic Alias Table

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.16

Related Version A full version [17] of the paper is available at `http://arxiv.org/abs/2005.03584`.

Supplementary Material Implementations and data: `https://ae.cs.uni-frankfurt.de/r/p/pps`.

Funding This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2, ME 2088/4-2, and ME 2088/5-1.

Acknowledgements This project was initiated on a workshop of the DFG FOR 2975/1. We thank the anonymous reviewers for their insightful comments and pointers, as well as the Center for Scientific Computing, University of Frankfurt, for making their HPC facilities available.

28th Annual European Symposium on Algorithms (ESA 2020).
Editors: Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders; Article No. 16; pp. 16:1–16:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

We consider the *population model*, introduced by Angluin et al. [5] to model systems of resource-limited mobile agents that interact to solve a common task. Agents are modeled as finite-state machines. The computation of a *population protocol* is a sequence of pairwise interactions of agents. In each interaction, the two participating agents observe each other's states and update their own state according to a transition function common to all agents.

Typical applications of population protocols are networks of passively mobile sensors [5]. As an example, consider a flock of birds, where each bird is equipped with a simple sensor. Two sensors communicate whenever their birds are sufficiently close. An application could be a distributed disease monitoring system raising an alarm if the number of birds with high temperature rises above some threshold. Further processes which resemble properties of population protocols include chemical reaction networks [38], programmable chemical controllers at the level of DNA [22], or biochemical regulatory processes in living cells [21].

While the computational power of population protocols with constantly many states per agent is well understood by now (see below), less is known about the power of protocols with state spaces growing with the population size. In this setting, much interest has been on analyzing the runtime and state space requirements for *probabilistic* population protocols, where the two interacting agents are sampled in each time step independently and uniformly at random from the population. This notion of a probabilistic scheduler allows the definition of a *runtime* of a population protocol. The runtime and the number of states are the main performance measures used in the theoretical analysis of population protocols.

For the theoretical analysis of population protocols, a large toolkit is available in the literature. Consequently, the remaining gaps between upper and lower bounds for many quantities of interest have been narrowed down: for many protocols, the required number of states has become sub-logarithmic, while the runtime approaches more and more the (trivial) lower bounds for any meaningful protocol. So far, when designing new protocols, simulations have always proven a versatile tool in getting an intuition for these stochastic processes. However, once observables are of order $\log\log n$ and below, naive population protocol simulators fail to deliver the necessary insights (e.g., $\log\log n \leq 5$ for typical input sizes of $n \leq 2^{32}$). Our main contribution in this paper is a new simulation approach allowing to execute a large number of interactions even if the population size exceeds 2^{40}. In the remainder of this section, we first give a formal model definition in Section 1.1 and then describe our main contributions and related work in Sections 1.2 and 1.3.

1.1 Formal Model Definition

In the *population model*, we are given a distributed system of n agents modeled as finite-state machines. A *population protocol* is specified by a state space $Q = \{q_1, \ldots, q_{|Q|}\}$, an output domain Y, a transition function $\delta\colon Q \times Q \to Q \times Q$, and an output function $\gamma\colon Q \to Y$. At time t, each agent i has a state $s_i(t) \in Q$, which is updated during the execution of the protocol. The current output of agent i in state $s_i(t)$ is $\gamma(s_i(t))$. The *configuration* $C(t) = \{s_1(t), \ldots, s_n(t)\}$ of the system at time t contains the states of the agents after t interactions. For the sake of readability, we omit the parameter t in $C(t)$ and $s_i(t)$ when it is clear from the context. The *initial configuration* is denoted $C(0) = C_0$.

The *computation* of a population protocol runs in a sequence of discrete time steps. In each time step, a *probabilistic scheduler* selects an ordered pair of agents (u, v) independently and uniformly at random to *interact*. Agent u is called the *initiator* and agent v is the *responder*. During this *interaction*, both agents u and v observe each other's state and update their states according to the transition function δ such that $(s_u(t{+}1),\ s_v(t{+}1)) \leftarrow \delta(s_u(t),\ s_v(t))$.

Table 1 Simulating N interactions among n agents in $|Q|$ states. For MULTIBATCHED, we restrict $|Q| = \omega(\sqrt{\log n})$. Values indicated by † hold in expectation.

	Simulator	Section	Time Complexity	Space Complexity (bits)
Sequential	$\text{SEQ}_{\text{Array}}$	Section 2	$\Theta(N)$	$\Theta(n \log \lvert Q\rvert)$
	$\text{SEQ}_{\text{Linear}}$	Section 2	$\mathrm{O}(N\lvert Q\rvert)$	$\Theta(\lvert Q\rvert \log n)$
	SEQ_{BST}	Section 2	$\Theta(N \log \lvert Q\rvert)$	$\Theta(\lvert Q\rvert \log n)$
	$\text{SEQ}_{\text{Alias}}$	Section 2	$\Theta(N)$ w.h.p.	$\Theta(\lvert Q\rvert \log n)$
Batch	BATCHED	Section 4	$\mathrm{O}(N(\log n + \lvert Q\rvert^2)/\sqrt{n})^\dagger$	$\Theta(\lvert Q\rvert \log n)$
	MULTIBATCHED	Section 5	$\mathrm{O}(N\lvert Q\rvert\sqrt{\log n}/\sqrt{n})^\dagger$	$\Theta(\lvert Q\rvert \log n)$

A given problem for the population model specifies the agents' initial states, the output domain O, and (a set of) *desired (output) configurations* for a given input. As an example, consider the MAJORITY problem. Each agent is initially in one of two states q_A and q_B corresponding to two opinions A and B. Assuming that A is the initially dominant opinion, the protocol concludes once all agents u give $\gamma(s_u) = A$ as their output. Any configuration in which all agents output the initially dominant opinion is a desired configuration.

This notion of a desired configuration allows to formally define two notions of a *runtime* of a population protocol. The *convergence time* T_C is the number of interactions until the system enters a desired configuration and never leaves the desired configurations in a given run. The *stabilization time* T_S is the number of interactions until the system enters a desired *stable* configuration for which there does not exist any sequence of interactions due to which the system leaves the desired configurations. A population protocol is *stable*, if it always eventually reaches a desired output configuration.

A number of variants of this model are commonly used. For *symmetric protocols*, the order of the interacting agents is irrelevant for the transition. In particular, this means that if $\delta(q_u, q_v) = (q'_u, q'_v)$, then $\delta(q_v, q_u) = (q'_v, q'_u)$. In *protocols with probabilistic transition functions*, the outcome of an interaction may be a random variable. In *one-way protocols*, only the initiator updates its states such that $\delta(q_u, q_v) = (q'_u, q_v)$ for any interaction.

Model Assumptions. We assume a *meaningful* protocol which converges after at most $N = \text{poly}(n)$ interactions, has an $\mathrm{O}(1)$ time transition function δ, and uses $|Q| < \sqrt{n}$ states (observe that many relevant protocols only use $|Q| = \mathrm{O}(\text{polylog}\, n)$ states; see Section 1.3).

1.2 Our Contributions

In this paper, we present a new approach for simulating population protocols. Our simulator allows us to efficiently simulate a large number, N, of interactions for large populations of size n. Our findings are summarized in Table 1.

Sequential Simulators. As a baseline, we directly translate the population model into a sequential algorithm framework SEQ: SEQ selects for each interaction two agents uniformly at random, updates their states, and repeats. We analyze the runtime and memory consumption of various variants in Section 2.

Batch Processing. To speed up the simulation, we introduce and exploit *collision-free runs*, a sequence of interactions where no agent participates more than once. Our algorithms BATCHED and MULTIBATCHED coalesce these independent interactions into batches for

improved efficiency. BATCHED first samples the length ℓ of a collision-free run. It then randomly pairs ℓ independent agents, adds one more interaction – the collision – reusing one of the run's agents, and finally repeats. BATCHED is presented in Section 4 and extended into MULTIBATCHED in Section 5. We discuss practical details and heuristics in Section 6.

Dynamic Alias Tables. The simulation of population protocols often needs an *urn-like* data structure to efficiently sample random agents (marbles) and update their states (colors). The *alias method* [41, 40] enables random sampling from arbitrary discrete distributions in $O(1)$ time. However, it is static in that the distribution may not change over time. Thus, we extend it and analyze a *Dynamic Alias Table* in Section 2. It supports sampling with and without replacement uniformly at random (u.a.r.) and addition of elements (if the urn is sufficiently full). Due to its practical performance and simplicity compared to more general solutions [30, 33], we believe this data structure might be of independent interest and show:

▶ **Theorem 1.** *Let U be a Dynamic Alias Table that stores an urn of n marbles, where each marble has one of k possible colors. U requires $\Theta(k \log n)$ bits of storage. If $n \geq k^2$, we can*

- *select a marble u.a.r. from U with replacement in expected constant time,*
- *select a marble u.a.r. from U without replacement in expected amortized constant time,*
- *and add a marble of a given color to U in amortized constant time.*

1.3 Related Work

Population Protocols. The population model was introduced in [5], assuming a constant number of states per agent. Together with [6, 9], they show that all semilinear predicates are stably computable in this model. In the following, we focus on two prominent problems, MAJORITY and LEADER ELECTION. For a broad overview, we refer to surveys [12] and [26].

In [8] a MAJORITY protocol with three states is presented where the agents agree on the majority after $O(n \log n)$ interactions w.h.p. (*with high probability* $1 - n^{-\Omega(1)}$), if the initial numbers of agents holding each opinion differ by at least $\omega(\sqrt{n} \log n)$. In [34, 25], four-state protocols are analyzed that stabilize in expectation in $O(n^2 \log n)$ interactions. In a recent series of papers [35, 1, 2, 4, 19, 15, 14, 13], bounds for the MAJORITY problem have been gradually improved. The currently best known protocol [13] solves MAJORITY w.h.p. in $O(n \log^{3/2} n)$ interactions using $O(\log n)$ states. Regarding lower bounds, [1] shows that protocols with less than $(\log \log n)/2$ states require in expectation $\Omega(n^2/\operatorname{polylog}(n))$ interactions to stabilize. In [2] it is shown that any MAJORITY protocol that stabilizes in $n^{2-\Omega(1)}$ expected interactions requires $\Omega(\log n)$ states under some natural assumptions.

The goal for LEADER ELECTION protocols is that exactly one agent is in a designated leader state. Doty and Soloveichik [24] show that any population protocol with a constant number of states that stably elects a leader requires $\Omega(n^2)$ expected interactions, a bound matched by a natural two-state protocol. Upper bounds for protocols with a non-constant number of states per agent were presented in [3, 1, 19, 2, 18, 28, 29, 16]. In [28] a LEADER ELECTION protocol that stabilizes w.h.p. in $O(n \log^2 n)$ interactions, using $O(\log \log n)$ states (matching a corresponding lower bound [1]) is presented. The core idea is to synchronize the agents using a *phase-clock*. The currently best known protocol for LEADER ELECTION is due to [16], stabilizing in expected $O(n \log n)$ interactions using $O(\log \log n)$ states per agent.

As a tool for self-synchronization, so-called *phase-clocks* have been explored in a wide range of related areas, see, e.g., the seminal paper [10]. In the population model, the concept of phase-clocks was first introduced in [7] under the assumption that a leader is present. These clocks were generalized in [28] to a *junta* of n^ε agents. In Section 7 we empirically analyze a variant of this phase-clock process.

Simulations have proven a versatile tool to get an intuitive understanding of population protocols. This is also reflected in the related work: See, e.g., [7, 8, 4, 3] for some examples of papers that also present empirical data. However, to the best of our knowledge, our paper is the first systematic analysis of simulators for population protocols.

Sampling from Discrete Distributions. The methods to sample non-uniform variates heavily depend on the modeling and properties of the required probability distributions.

If the distribution is governed by a closed-form density function $f(x)$, "numerical tricks" can yield efficient algorithms with small memory footprints; this is the case for most well-known distributions [23, 20]. A standard approach is the *inverse sampling technique* [23]. It needs to compute the inverse F^{-1} of f's cumulative density function $F(x) = \int_{-\infty}^{x} f(y)\mathrm{d}y$ (cf. Section 6.1). Another concept is *rejection sampling* [20, Sec. 5.2.5/6]. It obtains a sample x from a suitable simpler distribution g. In order to generate the distribution f, the sample is accepted only with probability proportional to $f(x)/g(x)$. The process repeats until a sample is obtained (cf. Section 3). A special case is the *ratio-of-uniforms* method, commonly used to obtain hypergeometric random variates [39].

Dedicated data structures support sampling from arbitrary discrete distributions. Given a finite universe $U = \{1, \ldots, u\}$ with probabilities $(p_1, \ldots, p_u)$ with $\sum_i p_i = 1$, Walker's *alias tables* [41] allow sampling in constant expected time, and can be constructed in $\mathrm{O}(u)$ time [40]; recently Hübschle-Schneider et al. [31] discussed engineering aspects and parallel construction.

In this paper, we model an urn with n balls with $\mathrm{O}(\sqrt{n})$ colors (typically much less) as a distribution over k colors where p_i is proportional to the number of balls with color i. While alias tables work in the static case, they do neither support removal nor insertion of balls without rebuilding the data structure.

A suited binary tree can be constructed in $\mathrm{O}(k)$ time, and supports updates and sampling in $\mathrm{O}(\log k)$ time. Two independent but similar data structures by [30] and [33] support updates and sampling in expected constant time. They partition the input into groups such that the probabilities of elements within a group differ by at most a factor of two. This allows rejection sampling *within* a group with an acceptance rate of at least 1/2. Since updates to the distribution can affect the partition sizes, over-provisioning and table doubling involving garbage collection [33, App. B] is used.

The approaches differ in the way they select a group to sample from. They are however both recursive in the sense that the group selection is carried out with another instance of the data structure itself. To achieve constant access times, the recursion is stopped after constantly many layers, and the remaining very small problem is treated as a special case. As a result, the data structures are quite complex, and were excluded in preliminary experiments due to performance considerations. By constraining the supported distributions, simpler schemes can be obtained [36, 27]. However, in our use case, they incur impractically high rejection rates, as we cannot give non-trivial bounds on the number of balls per color.

It is note-worthy that Hagerup et al. [30] operate on integer weights (modeled as *generalized distributions* lacking the normalization constraint), and require integer arithmetic only.

2 Sequential Simulation

As a baseline, we first consider variants of Seq, a sequential approach defined in Algorithm 1. It is a direct translation of the machine model discussed in Section 1.1. Seq carries out N steps in a fully serialized manner. For each interaction, it selects two agents uniformly at random, computes their new states based on their current ones, and updates the configuration.

Algorithm 1 SEQ: The algorithmic framework for sequential simulation.

input: configuration C, transition function δ, number of steps N
for $t \leftarrow 1$ **to** N **do**
  sample and remove agents i and j without replacement from C
  add agents in states $\delta(s_i, s_j)$ to C

Under the realistic assumption that the transition function δ can be evaluated in constant time, SEQ's runtime and memory footprint is dominated by storing, sampling from, and updating the configuration C. We therefore consider appropriate data structures. In the population model, agents typically are anonymous, i.e., we cannot distinguish two agents in the same state. Hence, we can store a configuration C as an unordered multiset $\hat{C}$ and maintain multiplicities rather than individual states. To this end, SEQ requires an *urn-like* data structure which efficiently supports (i) weighted sampling (with and without replacement) and (ii) adding of single agents. In the following, we consider various data structures and their impact on the complexity of the sequential approach.

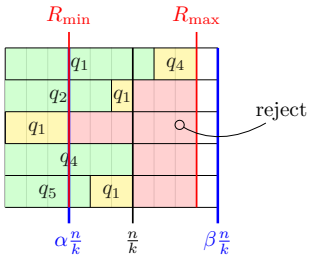
Array. $\text{SEQ}_{\mathsf{Array}}$ maintains the configuration C in an array $A[1 \dots n]$ where $A[i]$ holds s_i, the state of the i-th agent. Sampling with replacement is trivial, as we only draw a uniform variate $X \in [n]$ and return $A[X]$. Sampling without replacement works analogously: we overwrite $A[i]$ with $A[n]$ and remove the array's last element $A[n]$. Adding new elements is possible by appending. (Note that we do not grow the memory since we always store at most n agents in the array.) This leads to an $\mathrm{O}(N)$ time algorithm and a memory footprint of $\mathrm{O}(n \log |Q|)$ bits, which can be prohibitively large if simulating large populations in parallel.

Linear Search. $\text{SEQ}_{\mathsf{Linear}}$ maintains the multiset $\hat{C}$ in an array A such that $A[i]$ holds the number of agents in state q_i. Sampling requires a linear search on A in $\mathrm{O}(|Q|)$ per sample. This results in a worst-case simulation time of $\Theta(N|Q|)$. Nevertheless, in practice $\text{SEQ}_{\mathsf{Linear}}$ is among the fastest sequential variants for small $|Q|$ (see Section 7). Compared to $\text{SEQ}_{\mathsf{Array}}$, it has a significantly smaller memory footprint of $\mathrm{O}(|Q| \log n)$ bits.

Binary Search Tree. $\text{SEQ}_{\mathsf{BST}}$ maintains the multiset $\hat{C}$ using a balanced binary search tree. The i-th leaf (from left to right) encodes $\hat{C}_i$, the number of agents in state i. Each inner node v stores the number ℓ_v of agents in its left subtree. To randomly sample an agent, we draw an integer X from $\{0, \dots, n-1\}$ uniformly at random and compare it to the root's value ℓ_r. If $X < \ell_r$, the sample is in the interval covered by the left sub-tree, and we descend accordingly. Otherwise, we update $X \leftarrow X - \ell_r$ and descend into the right subtree. We recurse until some leaf i is reached, where we emit an agent of state i.

Each operation on the tree involves a simple path from the root to a leaf of length $\Theta(\log |Q|)$. Since the work per level is constant, all operations take $\Theta(\log |Q|)$ time. Thus, $\text{SEQ}_{\mathsf{BST}}$ requires $\Theta(N \log |Q|)$ total time and $\mathrm{O}(|Q| \log n)$ bits of memory.

$\text{SEQ}_{\mathsf{Alias}}$ combines the linear runtime of $\text{SEQ}_{\mathsf{Array}}$ (w.h.p.) with the small memory footprint of $\text{SEQ}_{\mathsf{BST}}$, provided $|Q| < \sqrt{n}$. At the heart of $\text{SEQ}_{\mathsf{Alias}}$ lies a *Dynamic Alias Table* introduced in the following section.



$F[i]$	$S[i]$	$A[i]$	rejection probability
7	2	q_4	0
5	1	q_1	$3/9$
0	3	q_1	$6/9$
6	0	q_1	$3/9$
4	2	q_1	$3/9$

Figure 1 Dynamic Alias Table storing $\hat{C} = (q_1 : 13,\ q_2 : 5,\ q_3 : 0,\ q_4 : 8,\ q_5 : 4)$, i.e., $n = 30$ and $k = 5$. This imbalanced configuration will soon need rebuilding, e.g., after the next decrease of q_1 in row 3, or after adding two more agents in state q_1 in row 1.

3 Dynamic Alias Tables

Dynamic Alias Tables. In this section we introduce our Dynamic Alias Table data structure. Our goal is to model an urn which initially contains n marbles, each of which has one of k possible colors. We assume that the colors are identified by numbers in $\{1, \ldots, k\}$. The urn defines a probability distribution D for the color of a marble drawn uniformly at random: let p_i be the probability that we sample a marble of color i.

Original Alias Method. The alias method [41] uses a table with two columns and k rows, one row for each element (color) in the distribution D. Each row i has two entries corresponding to two elements. Each element has a weight in $[0, 1]$, and the two weights sum up to 1 in each row. The first element of row i is always element i. It has weight $F[i]$. The second element of row i is stored in $A[i]$. It has a weight of $1 - F[i]$. This means that the original alias method uses only the two arrays, F and A, to store the distribution $p_1, \ldots, p_k$.

To sample from D in the original alias method, we first sample a row i uniformly at random from $\{1, \ldots, k\}$. Then we draw a random real $X \in [0, 1)$. If $X < F[i]$, we return the left element, i. Otherwise, we return the right element, $A[i]$. In the following, we modify the alias method and call the resulting data structure Dynamic Alias Table.

Dynamic Alias Tables. Recall that we assume that our distribution corresponds to an urn storing n marbles of k different colors. First, we explicitly add a second weight array $S[i]$ which stores the weight of the alias. Now instead of storing just one real value $F[i]$ for each row i, we store the exact numbers of marbles as integers for the first and the second entry of row i in $F[i]$ and $S[i]$, respectively (cf. generalized distributions of [30]). As before, the first entry of row i corresponds to color i and the second entry of row i corresponds to color $A[i]$. As illustrated in Figure 1, the rows are constructed in such a way that the total weight of each row no longer adds up to the real value 1, but to the integer value $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ such that all rows in total add up to n.

▶ **Observation 2.** *Let U be a Dynamic Alias Table encoding an urn with n marbles and k colors. The data structure U can be constructed in $O(k)$ time.*

Proof. The algorithm by Vose [40] can generate an (original) alias table representation of such a discrete probability distribution D in $O(k)$ time. It is straightforward to define a mapping between the weights of the original alias method as computed in [40] and the two integer values used in our Dynamic Alias Table. It follows that the Dynamic Alias Table (using integer weights) can be constructed in $O(k)$ time. ◀

Updating and Sampling from Dynamic Alias Tables. Our modified data structure now allows us to sample elements with and without replacement. Let $R_i = F[i] + S[i]$ denote the weight of row i and define $R_{\min}$ and $R_{\max}$ as smallest and largest row weights, respectively. Observe that in general $R_{\min} \neq R_{\max}$.

In order to sample from U, we first select row i uniformly at random from $\{1, \ldots, k\}$. Then we draw a uniform variate X from $\{0, \ldots, R_{\max} - 1\}$.[1] There are three possible events: If $X < F[i]$, we emit the first element i. If $F[i] \leq X < R_i$, we emit the second element $A[i]$. Otherwise, we reject the trial and restart the sampling process.

If we sample from U without replacement, we decrement the weight of the element just sampled. This is always possible, since only elements with strictly positive weights can be sampled in the first place. If we add a new element with color i to U, we increment the weight of the first element of row i.

In order to guarantee expected constant sampling time, we ensure that the fraction between $R_{\min}$ and $R_{\max}$ does not exceed a certain value. Let $0 < \alpha < 1$ and $\beta > 1$ be two parameters chosen such that $\beta/\alpha = \mathrm{O}(1)$. After each update to U we require

$$\alpha\lfloor n/k \rfloor \leq R_{\min} \leq R_{max} \leq \beta\lceil n/k \rceil. \tag{1}$$

Otherwise, we rebuild the data structure in $\mathrm{O}(k)$ time.

We are now ready to show Theorem 1.

Proof of Theorem 1. We start with the memory complexity. The Dynamic Alias Table U stores the values of k, n, and $R_{\max}$ as well as three arrays. Array $F[1 \ldots k]$ stores the weight of the first column, array $S[1 \ldots k]$ stores the weight of the second column, and array $A[1 \ldots k]$ stores the alias, i.e., the element of the second column. All entries are integers from $\{0, \ldots, n\}$ (recall that we assume $k \leq \sqrt{n}$). Thus, the Dynamic Alias Table requires $\Theta(k \log n)$ bits of memory.

Let us now consider the sampling procedure. First, we consider the rejection probability. Recall that we first sample a row i and then draw a uniform variate X from $\{0, \ldots, R_{\max} - 1\}$. As before, we denote the total weight of row i as R_i with $R_i = F[i] + S[i]$. A sampling trial in row i is rejected if $X \geq R_i$, i.e., with probability $R_i/R_{\max}$. Therefore, the probability to reject a sample from any row is at most $R_{\min}/R_{\max}$. From the conditions in Equation (1) we get that the rejection probability is at most $\alpha/\beta = \mathrm{O}(1)$ and, conversely, we have at least a constant success probability of $(\beta - \alpha)/\beta$. The number of trials until we emit an element is therefore geometrically distributed and has an expected value of at most $\beta/(\beta - \alpha) = \mathrm{O}(1)$.

It remains to show that we emit an element of color i with probability $p_i = \hat{C}_i/n$, where $\hat{C}_i$ is the number of marbles of color i in the Dynamic Alias Table U. We consider a single sampling trial. Observe that in each trial we are given a uniform probability space $\mathbf{\Omega} = \{(i, x) \ : \ 1 \leq i \leq k \text{ and } 0 \leq x < R_{\max}\}$. From this probability space we draw the row i and the value X uniformly at random. Fix a color c and let $\mathcal{S}_c$ be the set of all events (i, x) which lead to emission of an element of color c for this probability space $\mathbf{\Omega}$. An event (i, x) is in $\mathcal{S}_c$ if and only if (i) $i = c$ and $x < F[i]$ or (ii) $A[i] = c$ and $F[i] \leq x < F[i] + S[i]$.

The Dynamic Alias Table U is constructed such that the total weight for each color c always equals $\hat{C}_c$. Therefore, counting all elementary events gives us $|\mathcal{S}_c| = \hat{C}_c$. Observe that $\mathbf{\Omega}$ is a uniform probability space since the row i and the value X are drawn uniformly. It has size $|\mathbf{\Omega}| = kR_{\max}$. Hence, all events in $\mathcal{S}_c$ have equal probability $1/(kR_{\max})$, and we get $\Pr[\mathcal{S}_c] = |\mathcal{S}_c|/(kR_{\max}) = \hat{C}_c/(kR_{\max})$.

[1] Observe that in a practical implementation one can draw a single uniform variate U' from $\{1, \ldots, k \cdot R_{\max} - 1\}$, and derive U and X from it.

Let $\mathcal{R}$ be the event that a trial is rejected. Analogously to before, we enumerate over all elementary events and obtain $|\mathcal{R}| = \sum_i (R_{\max} - R_i) = kR_{\max} - n$. For the complementary event $\overline{\mathcal{R}}$ we get $|\overline{\mathcal{R}}| = n$, which matches the intuition that the urn contains n marbles. Hence, we have $\Pr[\overline{\mathcal{R}}] = n/(kR_{\max})$. Observe that $\mathcal{S}_c$ and $\mathcal{R}$ are mutually exclusive and hence $\mathcal{S}_c \cap \overline{\mathcal{R}} = \mathcal{S}_c \setminus \mathcal{R} = \mathcal{S}_c$.

Rejected trials emit no element, but are repeated. Hence, we condition on $\overline{\mathcal{R}}$ and obtain

$$p_c = \Pr[\mathcal{S}_c \mid \overline{\mathcal{R}}] = \frac{\Pr[\mathcal{S}_c \cap \overline{\mathcal{R}}]}{\Pr[\overline{\mathcal{R}}]} = \frac{\Pr[S_c]}{\Pr[\overline{\mathcal{R}}]} = \frac{\hat{C}_c}{kR_{\max}} \cdot \frac{kR_{\max}}{n} = \frac{\hat{C}_c}{n}.$$

This means that a color c is indeed emitted with the correct probability $p_c = \hat{C}_c/n$.

Finally, we consider the amortized costs of rebuilding the Dynamic Alias Table U ever so often. Observe that U has to be rebuilt whenever the condition in Equation (1) is violated. This can happen in two possible ways.

- Case 1: $R_{\min} < \alpha\lfloor n/k \rfloor$.
  In this case there must exist a row i for which $R_i < \alpha\lfloor n/k \rfloor$. Observe that by assumption of the theorem we have $n \geq k^2$, and after rebuilding U we have $R_{\min} = \lfloor n/k \rfloor$. In order to have $R_i < \alpha\lfloor n/k \rfloor$, at least $\lfloor n/k \rfloor - \alpha\lfloor n/k \rfloor \geq k(1-\alpha)$ elements must have been deleted from row i. As $0 < \alpha < 1$, this happens only after at least $k(1-\alpha) = \Omega(k)$ sampling operations. Now according to Observation 2, rebuilding takes time $\mathrm{O}(k)$. Together this implies that rebuilding U takes amortized constant time per update of U.
- Case 2: $R_{\max} > \beta\lceil n/k \rceil$.
  The second case follows analogously to the first case. If $R_{\max} > \beta\lceil n/k \rceil$, at least $\beta\lceil n/k \rceil - \lceil n/k \rceil \geq k(\beta-1)$ elements must have been added. This takes at least $k(\beta-1) = \Omega(k)$ insertions, and hence rebuilding U takes amortized constant time per insertion.

Removal and insertion operations can be arbitrarily mixed and interact only beneficially towards the amortization arguments. This concludes the proof of the theorem. ◀

4 Batch Processing

So far, we discussed algorithms to simulate a population protocol step-by-step. These simulators can output the population's configuration $C(t)$ for each time step $1 \leq t \leq N$. With a time complexity of $\mathrm{O}(N)$, the simulators $\textsc{Seq}_{\mathsf{Array}}$ and $\textsc{Seq}_{\mathsf{Alias}}$ are optimal in this sense. In practice, however, it often suffices to obtain a configuration snapshot every $\Theta(n)$ steps. In this setting, we can achieve sub-constant work per interaction under mild assumptions.

Recall that $\textsc{Seq}_{\mathsf{BST}}$ has a small memory footprint but a sub-optimal time complexity of $\Theta(N \log |Q|)$. Observe, however, that the underlying binary search tree can update the multiplicity of any existing state in time $\mathrm{O}(\log |Q|)$ *independently* of the changed quantity. Here, we introduce the new algorithm Batched (see Algorithm 2) to exploit this observation. The algorithm uses a binary search tree to store the configuration. It updates $\Omega(\sqrt{n})$ agents in expectation with each access and therefore reduces the time complexity to $\mathrm{O}(N(\log n + |Q|^2)/\sqrt{n})$ which is $\mathrm{o}(N)$ for $|Q| = \mathrm{o}(n^{1/4})$ and $N = \Theta(\mathrm{poly}(n))$.

Batching interactions. In order to coalesce individual updates into batches, Batched uses the notion of *collision-free runs* as illustrated in Figure 2. We interpret the execution of a protocol as a sequence $i_1, i_2, \ldots$ where at time t agents i_{2t-1} and i_{2t} interact.

Let ℓ be the largest index such that all $i_1, \ldots, i_\ell$ are distinct. Then, the first $\lfloor \ell/2 \rfloor$ interactions are independent of each other and can be rearranged in any order. We refer to them as a collision-free run of length ℓ. If ℓ is odd, the first agent of the $(\lfloor \ell/2 \rfloor + 1)$-th

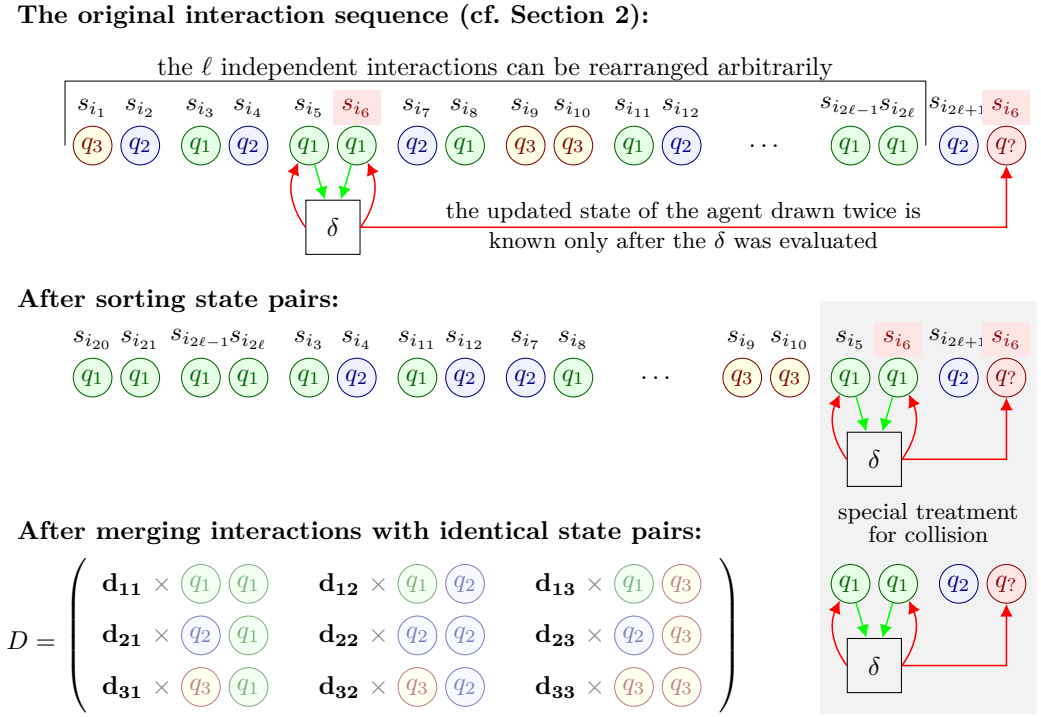


Figure 2 Batch processing uses collision-free runs, long sequences of independent interactions, which can be rearranged and grouped together.

interaction is also considered collision-free. Since we are free to reorder the interactions, we can group all interactions of states (q_i, q_j) together, evaluate $\delta(q_i, q_j)$, and update all accordingly affected states in one step.

Now instead of sampling a sequence of agents and partitioning the sequence into collision-free runs, we take the opposite direction. We first sample only the length ℓ of a collision-free run from the appropriate probability distribution (see below). Then, we randomly match ℓ agents as discussed below. Finally, we reuse one of the agents from the matching in order to plant a collision. These steps are repeated until at least N interactions are simulated.

Matching Agents. We simulate sampling ℓ agents without replacement to construct a collision-free run of length ℓ. While we cannot afford to draw the agents individually, we only need to know how many interactions n_{ij} of each state pair (q_i, q_j) we encountered. Thus, a run can be modeled by a $|Q|\times|Q|$ matrix $D = (n_{ij})$ with $\sum_{ij} n_{ij} = \lfloor \ell/2 \rfloor$. (If ℓ is odd, we remove one agent and treat it individually.)

To obtain D, we first sample the row sums $D_i = \sum_j n_{ij}$ of the matrix from a multivariate hypergeometric distribution. This simulates sampling $\lfloor \ell/2 \rfloor$ initiating agents without replacement. We then sample values within each row analogously to find the matching responding agents. Sampling D takes $\mathrm{O}(|Q|^2)$ time in total since each individual sample from a hypergeometric distribution can be computed in $\mathrm{O}(1)$ time [39].

For correctness, note that our sampling approach corresponds to first selecting $\lfloor \ell/2 \rfloor$ agents as initiators and then $\lfloor \ell/2 \rfloor$ agents as responders. That is, we first sample agents $i_1, i_3, \ldots, i_{2\lfloor \ell/2 \rfloor - 1}$ and then agents $i_2, i_4, \ldots, i_{2\lfloor \ell/2 \rfloor}$ (instead of the natural interleaved variant $i_1, i_2, \ldots, i_{2\lfloor \ell/2 \rfloor}$). Since, each draw is taken uniformly at random, the permutation does not change the distribution (see full version [17] for a formal proof).

Algorithm 2 BATCHED: The algorithmic framework for simulation in batches.

input: configuration C, transition function δ, number of steps N
$t \leftarrow 0$
while $t < N$ **do**
  $\ell \leftarrow$ sample length of a collision-free run
  let $D = (d_{ij})$ be a $|Q| \times |Q|$ matrix and sample d_{ij} as the number of interactions (q_i, q_j) among ℓ interactions ▷ `batch processing`
  let C' be an empty configuration
  foreach $(q_i, q_j) \in Q^2$ **do**
    remove from C: d_{ij} agents in states q_i, and d_{ij} agents in states q_j
    $(q_i', q_j') \leftarrow \delta(q_i, q_j)$
    add to C': d_{ij} agents in states q_i', and d_{ij} agents in states q_j'
  if ℓ *is even* **then** ▷ `plant a collision`
    sample agent c_1 without replacement from C' ▷ `collision at` c_1
    merge C' into C
    sample agent c_2 without replacement from C
  else
    sample agent c_1 without replacement from C
    sample agent c_2 without replacement from C' ▷ `collision at` c_2
    merge C' into C
  add agents $\delta(c_1, c_2)$ to C
  $t \leftarrow t + \ell + 1$

Length of a Collision-Free Run. In the following, we analyze the length ℓ of a collision-free run. Observe that the following analysis is similar to the analysis of a generalized variant of the birthday problem [32]. We consider a generalization which we also use in Section 5. We assume that r agents have already interacted and ask how many more collision-free agents can be added. Formally we define the distribution $\mathsf{coll}(n, r)$ as follows.

▶ **Definition 3.** Consider a sequence $a_1, a_2, \ldots$ of agents sampled independently and uniformly at random. Let A_0 be a set of r initially prescribed agents and let $A_i = A_{i-1} \cup \{a_i\}$ be the set of agents after i draws. We define the random variable ℓ as the smallest index s.t. $a_\ell \in A_{\ell-1}$. We say $\ell \sim \mathsf{coll}(n, r)$, where n is the total number of agents and r is the number of prescribed agents.

In Section 6.1 we discuss how we can sample from this distribution using the inverse sampling technique [23]. We find that sampling ℓ takes $\mathrm{O}(\log n)$ time. In practice, this is comparable to the time it takes to sample a hypergeometric random variate. In the following, we show basic properties of $\mathsf{coll}(n, r)$ in order to show bounds on the runtime of BATCHED.

▶ **Lemma 4.** *Let $\ell \sim \mathsf{coll}(n, r)$. Then ℓ has support $\{1, \ldots, n - r\}$ and distribution* $\Pr[\ell = k] = n^{-(k+1)}(n - r)!(r + k)/(n - r - k)!$.

Proof. Consider an urn with n marbles. Initially, r marbles are red, while the remaining $n - r$ marbles are green. We now take out one marble at a time: if it is green, we keep on going (think of a traffic light) and put a red one back in. If we take a red marble, we stop. Observe that the number of marbles we take out is exactly ℓ as above, as the green marbles represent new unconsidered agents while the red ones represent agents in $A_{\ell-1}$.
This directly leads to the acclaimed distribution:

$$\Pr[\ell = k] = \underbrace{\prod_{i=0}^{k-1} \frac{(n-r)-i}{n}}_{\text{select } k \text{ out of } n-r} \cdot \underbrace{\frac{r+k}{n}}_{(k+1)\text{-th is red}}$$

◀

▶ **Lemma 5.** *Let $\ell \sim \mathsf{coll}(n, 0)$. Then $\mathbb{E}[\ell] = \Theta(\sqrt{n})$.*

Proof. We first upper bound $\mathbb{E}[\ell] = \mathrm{O}(\sqrt{n})$ and then give a matching lower bound $\mathbb{E}[\ell] = \Omega(\sqrt{n})$. In both cases, we write $\mathbb{E}[\ell] = \sum_{i=0}^{n} \Pr[\ell \geq i]$ and split the sum at $\sqrt{n}$. Then we bound both terms appropriately. Observe that for some fixed value i we have $\Pr[\ell \geq i] = \prod_{j=0}^{i-1}(1 - j/n)$. For the upper bound on $\mathbb{E}[\ell]$ we get

$$\mathbb{E}[\ell] = \sum_{i=0}^{n} \Pr[\ell \geq i] = \sum_{i=0}^{n} \prod_{j=0}^{i-1} \left(1 - \frac{j}{n}\right) \leq \sum_{i=0}^{\sqrt{n}-1} 1 + \sum_{i=\sqrt{n}}^{\infty} \left(1 - \frac{\sqrt{n}}{n}\right)^{i} \leq 2\sqrt{n}.$$

Similarly, we get for the lower bound on $\mathbb{E}[\ell]$ that

$$\begin{aligned}\mathbb{E}[\ell] &= \sum_{i=0}^{n} \Pr[\ell \geq i] = \sum_{i=0}^{n} \prod_{j=0}^{i-1} \left(1 - \frac{j}{n}\right) \geq \sum_{i=0}^{\sqrt{n}} \prod_{j=0}^{i-1} \left(1 - \frac{\sqrt{n}}{n}\right) = \sum_{i=0}^{\sqrt{n}} \left(1 - \frac{1}{\sqrt{n}}\right)^{i} \\ &= \sqrt{n}\left(1 - \left(1 - \frac{1}{\sqrt{n}}\right)^{\sqrt{n}+1}\right) \geq \sqrt{n}\left(1 - e^{-1}\right).\end{aligned}$$

Therefore we have $\mathbb{E}[\ell] = \Theta(\sqrt{n})$. ◀

Using Lemma 5, we are now ready to bound the runtime and space complexity of Batched.

▶ **Theorem 6.** *Let n be the number of agents and $|Q|$ the number of states. Batched simulates N interactions in $\mathrm{O}(N(|Q|^2 + \log n)/\sqrt{n})$ expected time using $\Theta(|Q| \log n)$ bits.*

Proof. According to Lemma 5, each batch simulates $\Theta(\sqrt{n})$ interactions in expectation. It takes $\mathrm{O}(\log n)$ time to sample the length of a collision-free run ℓ (see Section 6.1) and $\mathrm{O}(|Q|^2)$ time (cf. [39]) to sample the interaction numbers and process the interactions for all pairs of states. This implies the runtime complexity. The space complexity follows immediately from the binary search tree used to store the configuration. ◀

5 Merging Batches

In an empirical evaluation, we found that our implementation of algorithm Batched spends most time in the batch processing step (to sample and transition the $|Q| \times |Q|$ matrix D); this is especially true for complex protocols with non-trivial state space sizes. As the matrix sampling cost is independent of the length ℓ of the underlying collision-free run, we modify the algorithm to support more than one collision per batch processing step.

Introducing Epochs. An execution of the improved algorithm MultiBatched logically consists of several epochs. For each epoch, the algorithm samples the *lengths* $\ell_1, \ell_2, \ldots, \ell_\rho$ of multiple collision-free runs $R_1, \ldots, R_\rho$. As no agent may appear twice in the union of those collision-free sequences, later runs become shorter in expectation ($\mathbb{E}[\ell_{i+1}] < \mathbb{E}[\ell_i]$), naturally limiting the number ρ of runs per epoch. After each run R_i, we plant one collision, i.e., an interaction with an agent that was already considered in the current epoch. An epoch concludes with a single batch processing step, in which matrix D is sampled and processed analogously to algorithm Batched.

Tracking Dependencies. While algorithm Batched only reorders and groups together independent interactions, our improved algorithm MultiBatched delays most interactions until the end of the epoch. To do so, the algorithm conceptually assigns each agent one of three types, and updates these labels as it progresses through the epoch:

- untouched agents did not interact in the current epoch. Hence, all agents are labeled untouched at the beginning of an epoch.
- updated agents took part in at least one interaction that was already evaluated. Thus, updated agents are already assigned their most recent state.
- delayed agents took part in *exactly one* interaction that was not yet evaluated. Thus, delayed agents are still in the same state they had at the beginning of the epoch, but are scheduled to interact at a later point in time. We additionally require that their interaction partner is also labeled delayed.

Analogously to algorithm Batched, we maintain two urns C and C'. Urn C' contains updated agents, while urn C stores untouched and delayed agents (or in other words, all agents whose state was not updated in the current epoch). At any point in time, an agent is either in C or C' meaning that $|C| + |C'| = n$. Due to symmetry, we do not explicitly differentiate untouched from delayed agents. We rather maintain only the number T of delayed agents and lazily select them while planting collisions or during batch processing.

If a delayed agent a is selected while planting a collision, it takes part in a second interaction and – by definition – cannot be labeled delayed any more. Thus, we randomly draw a second delayed agent b, evaluate their transition, store the updated state of b in C', and directly evaluate a again in the planted collision. Finally, we decrease $T \leftarrow T - 2$ as agents a and b changed their labels from delayed to updated. Observe that we might repeat this step in the (unlikely) case that a planted collision involved two formerly delayed agents.

Length of an Epoch. We now analyze the length of an epoch. We start by extending the analysis of $\mathsf{coll}(n, r)$ to the $r = \Omega(\sqrt{n})$ regime (reached after $\mathrm{O}(1)$ runs w.h.p.). The following lemmas establish expected value and concentration.

▶ **Lemma 7.** *Let $\ell \sim \mathsf{coll}(n, r)$ and $r = \Omega(\sqrt{n})$. Then $\mathbb{E}[\ell] = \Theta(n/r)$.*

Proof. The proof follows analogously to Lemma 5. Again, we start with the upper bound.

$$\mathbb{E}[\ell] = \sum_{i=0}^{n-r} \Pr[\ell \geq i] = \sum_{i=0}^{n-r} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \leq \sum_{i=0}^{\infty} \left(1 - \frac{r}{n}\right)^i = \frac{n}{r}.$$

For the lower bound we derive a general result for arbitrary r.

$$\begin{aligned}\mathbb{E}[\ell] &= \sum_{i=0}^{n-r} \Pr[\ell \geq i] = \sum_{i=0}^{n-r} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \geq \sum_{i=0}^{r-1} \prod_{j=0}^{i-1} \left(1 - \frac{j+r}{n}\right) \geq \sum_{i=0}^{r-1} \left(1 - \frac{2r}{n}\right)^i \\ &= \frac{n}{2r}\left(1 - \left(1 - \frac{2r}{n}\right)^r\right) \geq \frac{n}{2r}(1 - e^{-2r^2/n}).\end{aligned}$$

The last inequality holds since $e^{-2r^2/n}$ constitutes an upper bound on $(1 - 2r/n)^r$ as it can be rewritten as $(1 - 2r/n)^{n \cdot r/n}$ and $(1 - 2r/n)^n \leq e^{-2r}$. For $r = \Omega(\sqrt{n})$ the second factor $(1 - \exp(-2r^2/n))$ is $\Omega(1)$ which proves the claim. ◀

▶ **Lemma 8.** *Let $\ell \sim \mathsf{coll}(n, r)$ and $r = \Omega(\sqrt{n})$. Then $\ell = \Theta(n/r)$ with probability $1 - o(1)$.*

Proof. We prove the claim by showing that $\Pr[\ell < t]$ and $\Pr[\ell > t]$ are $o(1)$ for $t = o(n/r)$ and $t = \omega(n/r)$, respectively. We have

$$\Pr[\ell < t] = 1 - \Pr[\ell \geq t] = 1 - \prod_{i=0}^{t-1}\left(1 - \frac{i+r}{n}\right) \leq 1 - \left(1 - \sum_{i=0}^{t-1}\frac{i+r}{n}\right) \leq \frac{2tr + t^2}{2n},$$

where the first inequality follows from the Weierstrass product inequality. Furthermore, with $r = \Omega(\sqrt{n})$ we have $n/r = O(\sqrt{n})$ and thus $t = o(n/r)$ such that $t^2 = o(n)$ and $tr = o(n)$. For the second part, we have

$$\Pr[\ell > t] = \prod_{i=0}^{t}\left(1 - \frac{i+r}{n}\right) \leq \left(1 - \frac{r}{n}\right)^t \leq e^{-rt/n},$$

and for $t = \omega(n/r)$ and thus $rt/n = \omega(1)$ the claim follows. ◀

Intuitively, Lemma 7 shows that for sufficiently many prescribed agents r, the probability of drawing a colliding agent remains approximately r/n throughout the run. Similar to a geometric distribution, this results in a concentrated expected length of $\Theta(n/r)$. We now estimate the number of agents sampled after ρ runs.

▶ **Lemma 9.** *Let $L_k = \sum_{i=1}^{k} \ell_i$ be the number of agents drawn in an epoch with k runs. Then, for $|Q| = \omega(\sqrt{\log n})$, $|Q| = o(\sqrt{n \log n})$ and $\rho = \mathrm{O}(|Q|^2/\log n)$ we have $\mathbb{E}[L_\rho] = \Theta(\sqrt{\rho n})$.*

Proof. The variable L_k equivalently corresponds to the number of marbles $B(k, n)$ that need to be drawn in the birthday problem s.t. k coincidences occur. The asymptotics of $\mathbb{E}[B(k, n)]$ have first been studied by Kuhn and Struik [32] for the cases that $k = o(n^{1/4})$. Their results have since been improved by Arratia et al. [11] where the asymptotic bounds on the moments of $B(k, n)$ have been calculated for more general conditions on k. By [11, Corollary 12] for k a function of n, i.e., $k = k_n$ where $k_n \to \infty$ and $k_n/n \to 0$ it holds that

$$\mathbb{E}[B(k_n, n)] \sim \sqrt{2nk_n} \quad \text{as } n \to \infty.$$

By assumption the conditions are met since $\rho = \omega(1)$ and $\rho = o(n)$, thus $\mathbb{E}[L_\rho] = \Theta(|Q|\sqrt{n/\log n}) = \Theta(\sqrt{\rho n})$. ◀

Complexity. In order to analyze MultiBatched's runtime, we first establish the time required per epoch, and then bound the total expected runtime and memory requirements.

▶ **Lemma 10.** *MultiBatched takes time $\mathrm{O}(\rho \log n + |Q|^2)$ for an epoch of ρ runs.*

Proof. Planting a collision is done by drawing the two interacting agents from the appropriate urns and setting them to be updated which requires $\mathrm{O}(1)$ time. Sampling the length of a single collision-free run takes time $\mathrm{O}(\log n)$ (see Section 6.1). For the final batch-processing step MultiBatched takes $\Theta(|Q|^2)$ time independently of the number of delayed agents. ◀

▶ **Theorem 11.** *Let n be the number of agents and $|Q|$ the number of states. MultiBatched simulates N interactions in $\mathrm{O}(N|Q|/\sqrt{n/\log n})$ expected time if $|Q| = \omega(\sqrt{\log n})$ and $|Q| = \mathrm{o}(\sqrt{n \log n})$.*

Proof. Combining Lemmas 9 and 10, we find a runtime of $\mathrm{O}(N(\rho \log n + |Q|^2)/\sqrt{\rho n})$. Setting $\rho = \Theta(|Q|^2/\log n)$ balances the cost of sampling runs and planting collisions with the cost of batch processing, and thus does not increase the asymptotic cost per epoch. Higher values of ρ only increase the expected time complexity. ◀

MultiBatched has sub-constant work per interaction for $|Q| = \mathrm{o}\left(\sqrt{\frac{n}{\log n}}\right)$ and $N = \Theta(\mathrm{poly}\, n)$.

6 Heuristics and Implementation Details

Implementations of all discussed simulators (including scripts to reproduce figures and numbers included in this paper) are freely available. In the following, we highlight important aspects necessary to obtain simulators that are both fast in practice and highly customizable.

All simulators are implemented in C++ and use compile-time specializations to implement specific protocols and experimental setups.

In contrast to pure deterministic functions, non-deterministic transition functions (possibly with side-effects) have to be informed about every interaction carried out.[2] To allow for batch processing, we cannot use the natural invocation order. Instead, we inform the protocol how often a state pair will interact within an epoch. It is then expected to assign all participating agents to the appropriate states. See the full version of this paper [17] for more details and listings.

6.1 Sampling the Length of a Collision-Free Run

Recall that Batched and MultiBatched repeatedly sample the length ℓ of a collision-free run. In the following we discuss how this sampling can be implemented using the inverse sampling technique (Ist) [23]: let $\mathrm{cdf}(x)$ be the cumulative density function of a target distribution. Then, Ist draws a uniform variate U from $[0;1]$, solves $U = \mathrm{cdf}(x)$ for x, and returns it as the sample. We denote the CDF of $\mathsf{coll}(n,k)$ as $F_{n,k}(t)$. Lemma 4 yields:

$$1 - F_{n,k}(t) = \Pr[\ell > t] = \prod_{i}^{t} \frac{n-k-i}{n} = \frac{1}{n^t} \frac{(n-k)!}{(n-k-t-1)!} \overset{(x-1)!=\Gamma(x)}{=} n^{-t} \frac{\Gamma(n-k+1)}{\Gamma(n-k-t)}.$$

Since we are not aware of an inverse that can be evaluated fast, we numerically solve $U = F_{n,k}(t)$ for t. To avoid numerical instabilities, we rewrite the expression in terms of $\log\Gamma(x)$, which is available as the C standard function `lgamma`(x):

$$U = 1 - n^{-t} \frac{\Gamma(n-k+1)}{\Gamma(n-k-t)} \quad \Leftrightarrow \quad \log(1-U) = \log\Gamma(n-k+1) - \log\Gamma(n-k-t) - t\log n$$

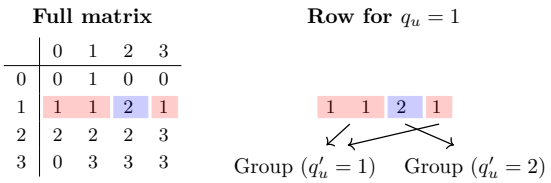
Lacking a cheap derivative of the RHS, we rely on first-order numerical inversion methods only. In this context, an ad-hoc combination of binary search and regula-falsi gave most consistent results. We jump-start the search using a small look-up table containing lower and upper bounds on t for intervals of U and k.

While the method requires $\mathrm{O}(\log n)$ evaluations of $F_{n,k}(\cdot)$, we observe less than ten calls on average for $n = 2^{50}$. The resulting sampling algorithm has a practical runtime comparable to the sampling of hypergeometric random variates. Since the latter is sampled much more frequently, further optimizations will yield limited results to the total runtimes of Batched and MultiBatched.

6.2 Heuristics

The frequent sampling of hypergeometric random variates, dominates MultiBatched's runtime. In the following, we discuss three heuristics to reduce this number.

[2] Observe that many protocols with non-deterministic transition functions have been derandomized, see, e.g., the notion of (biased) synthetic coins in [1, 18]. While this is supported by the simulator, we feel it is more convenient to offer the most expressive interface possible.



	0	1	2	3
0	0	1	0	0
1	1	1	2	1
2	2	2	2	3
3	0	3	3	3

Figure 3 Simplified transition matrix Δ' for a clock with period m=4. The initiator's phase (row) is circularly incremented only when matched with a suitable responder (column).

Renaming. In the *renaming heuristic* we exploit the observation that agents are typically not uniformly distributed over all states. Instead there are often sparsely populated states which are seldom hit when sampling an agent.

It can be beneficial to consider these states last. SEQ$_{\mathsf{Linear}}$'s linear search, for instance, stops as soon as the sampled state is found. The same is true when sampling BATCHED's and MULTIBATCHED's interaction matrices: for row i, we draw D_i agents from a multivariate hypergeometric distribution. This is implemented by obtaining $|Q|-1$ properly parametrized hypergeometric variates; the process terminates early once all D_i agents have been sampled.

In both examples, we maximize the probability of early stopping by considering highly populated states first. To this end, we maintain a permutation $\pi\colon [|Q|] \to [|Q|]$ that sorts states decreasingly by their sizes. We then process states in the order indicated by π. If this permutation is updated once every $\Omega(|Q| \log |Q|)$ interactions, the sorting step becomes asymptotically negligible for sequential simulators. For BATCHED and MULTIBATCHED, π can be updated once every $\Omega(\log q)$ batches.

Partitioning. If δ is a deterministic function, we can model it as a matrix $\Delta \in (Q \times Q)^{|Q|\times|Q|}$. The matrix of a deterministic one-way protocol can be further simplified to $\Delta' \in Q^{|Q|\times|Q|}$ since the states of the responders remain unchanged.

For many meaningful protocols, the entries of Δ' are not random but exhibit some structure. As an example, consider the simplified[3] phase-clock transition matrix illustrated in Figure 3. Here, each row contains only two different output states. The *partitioning heuristic* uses this observation during the batch steps of BATCHED and MULTIBATCHED. When sampling D_i responders for initiators in state q_i, we group together all entries in the i-th row of Δ' that assign the same new state to the initiating agents. It then suffices to draw one random hypergeometric variate per group.

Note that the heuristic does not reduce the runtime complexity of the algorithm – even if we precompute the partitioning. This is due the fact that we still need to compute the population sizes for each group which involves $\Theta(|Q|)$ additions per row. For pathological protocols, the number of hypergeometric random variates required for each row remains $\Omega(|Q|)$. A simple worst-case protocol is the transition function $\delta(q_u, q_v) = (q_v, q_v)$.

Skipping. Generalizing *partitioning heuristic* to two-way protocols tends to be ineffective in practice: since initiator and responder may both update their states, the transition matrix is often more fragmented. The partitioning overhead then easily exceeds the potential savings. For such protocols MULTIBATCHED uses a coarser partitioning, and only detects and skips transitions that preserve the configuration (i.e., $\delta(q_u, q_v) = (q_u, q_v)$ or $\delta(q_u, q_v) = (q_v, q_u)$).

[3] Formally the states of the phase-clock are pairs (x, b) where x represents the *phase*, and b marks an agent as *leader*. For the sake of simplicity we assume that all agents are followers.

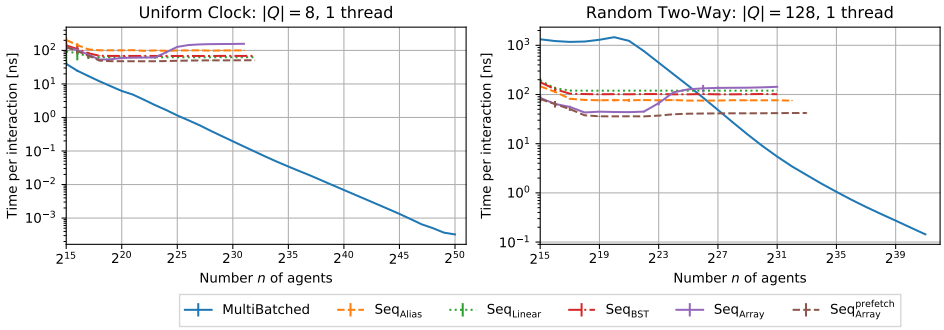


Figure 4 Processing time per interaction as function of the number of agents n. Each series ends with the largest value n for which the median of the total processing time is below 400 s.

6.3 Dynamic Epoch Lengths

Recall that MultiBatched is split into several epochs that each consist of multiple collision-free runs. In our implementation, we add runs to an epoch until the number of interactions exceeds a specified threshold T. The value of T has to be chosen as a trade-off between the cost of adding another run (i.e., sampling the run length and planting a collision) versus the diminishing return it yields (as later runs become shorter in expectation). This trade-off depends on the protocol and its configuration. For instance, the batch processing cost of a convergent protocol may become smaller compared to the initial costs (e.g., when most agents are in only a small fraction of the states).

As the trade-off is dynamic, we maximize the throughput using a control loop that dynamically optimizes the length of an epoch. Given the currently best value known for T, it increases (and later decreases) T to $1.1T$ and $0.9T$, respectively. For each of the three values, we measure the throughput, chose the T which maximizes it and repeat. Since the throughput response curve is single-peaked, the process will find a nearly optimal T.

7 Experimental Evaluation

In the following, we empirically evaluate the various simulation algorithms. The code is compiled using `g++-8.3` with flags `-O3 -march=native` and executed on the following system: 2× Intel Xeon Gold 6148 CPU @ 2.4 GHz (40 cores/80 hardware threads in total), 192 GiB DDR4 RAM @ 2666 MHz. Each data point is the median of at least five measurements (using different random seeds); error bars indicate their standard deviation.

Seq_{BST}'s search tree is implemented as an array with breath-first-indexing (i.e., the weight of node $i \geq 1$ is stored at $A[i]$; its left child is at index $2i$, its right child at $2i+1$). In order to reduce pipeline stalls, we implement a branch-free traversal similarly to [37]. $\text{Seq}_{\text{Array}}$ uses an array with 32 bit words to store states. We additionally consider $\text{Seq}_{\text{Array}}^{\text{prefetch}}$ which prefetches states for eight[4] interactions ahead of time as a latency hiding technique. If our Dynamic Alias Table implementation detects an imbalance necessitating rebuilding of the data structure, it will first attempt to quickly solve the problem by swapping the alias of the affected row with up to five randomly selected partners; only if this fails a rebuild is issued. This heuristic does not change the asymptotic time complexity of the data structure.

[4] This is the optimal value measured for this CPU type and slightly varies between machines.

The runtime of most simulators has non-trivial dependencies on the input parameters, protocol, and state distribution. Hence, we simulate a small number $N = n$ of interactions to prevent measuring artifacts caused by significant changes in the state distributions. MultiBatched typically simulates slightly more interactions due to the batching granularity. Since the runtime of all simulators is linear in N, we always report the time *per interaction* to ease extrapolation.

Three different protocols are used to highlight certain aspects of the algorithms.

- *Uniform Clock* and *Running Clock* implement the same deterministic one-way protocol phase-clock protocol inspired by [28]. In the *running* variant, all agents start in the first phase, and $\sqrt{n}$ of them are marked. Due to the choice of parameters, we expect only one out of $\Theta(\sqrt{n})$ interactions to change states. Thus, even at the end of the benchmark, the population is still highly concentrated in the lowest phase. The *uniform* variant, in contrast, evenly distributes marked and unmarked agents over all phases. This results in a constant update probability per interaction.
- The *Random Two-Way* protocol uses a deterministic transition function $\delta(q_i, q_j) = d_{ij}$ where each d_{ij} is initially drawn independently and uniformly at random from Q. Initially agents are evenly distributed over all states.

To ensure the correctness of our implementations, we rely on unit tests (e.g., using a family of protocols that count the number of interactions of each agent). Additionally, we cross-reference the results of various algorithms. Since we cannot expect two simulators to yield the same set of interactions, we compare their simulated dynamics. We, for instance, monitor the number of interactions required until a Uniform Clock starts running.

Number of Agents. We begin our experimental study by investigating the dependencies on the problem size n. To this end, we search for the largest number n of agents that a simulator can simulate within a fixed time budget of 400 s. Figure 4 reports such measurements for two different settings (see the full version of this paper [17] for the full set).

For the *Uniform Clock* protocol with $|Q| = 8$ states, the fastest Seq variant reaches $n = 2^{32}$ within the time budget. In the same time, MultiBatched simulates $N = n = 2^{50}$ interactions. For *Random Two-Way*, the ratio between the achievable population sizes is smaller but still exceeds three orders of magnitude. We attribute the different ratios mainly to the batching step. MultiBatched requires $\Theta(|Q|^2)$ hypergeometric variates per batching step for the latter protocol, since neither the partitioning nor the skipping heuristics (see Section 6) are effective on a featureless transition matrix with uniformly random states. For the *Uniform Clock* protocol, the partitioning heuristic reduces the number of hypergeometric random variates to less than $2|Q|$ per batch.

Number of States. As summarized in Table 1, the number $|Q|$ of states crucially affects the algorithms' runtimes. To quantify the practical impact, we carry out scaling experiments for $4 \leq |Q| \leq 128$ while fixing all remaining parameters. Figure 5 visualizes the results.

MultiBatched performs best in all cases supporting our previous analysis. It shows almost no scaling behavior for both clock protocols. For *Random Two-Way*, the algorithm is almost one order of magnitude faster than its competitors despite a slow-down of 40 between the smallest and largest state sizes.

$\text{Seq}^{\mathsf{prefetch}}_{\mathsf{Array}}$ is the second fastest solution in almost all settings. In the *Running Clock* campaign, it is however outperformed by $\text{Seq}_{\mathsf{Linear}}$. This can be explained by the fact that initially almost all agents are in state 0 which results in a constant time look-up despite the usage of a linear search. This behavior motivates the renaming heuristic.

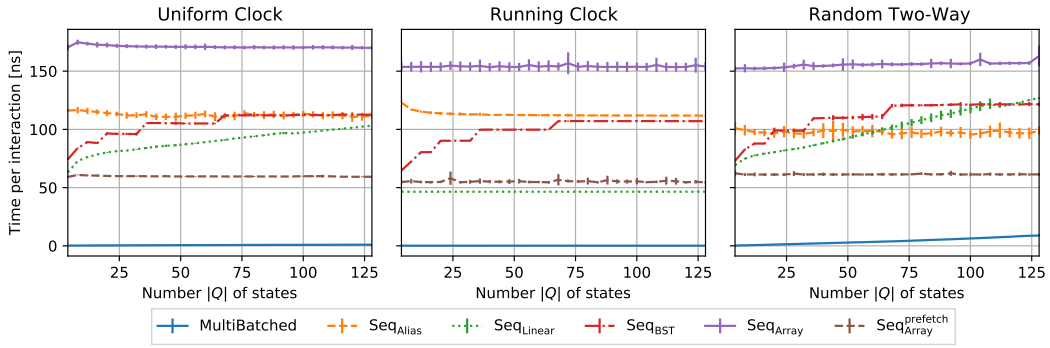


Figure 5 Processing time per interaction as function of the number of states $|Q|$ with $n = 2^{30}$.

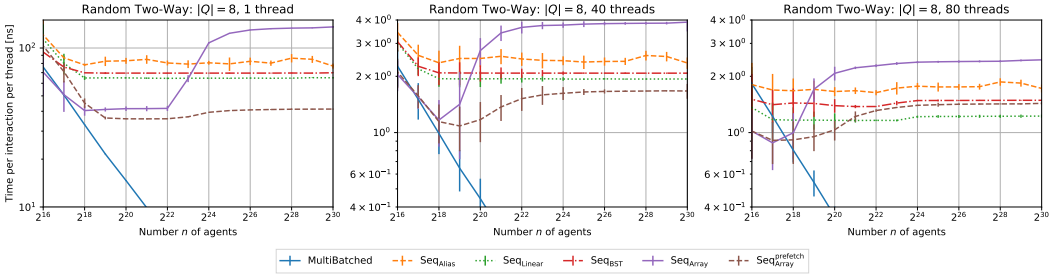


Figure 6 Effect of process parallelism on a machine with 40 CPU cores (plus HyperThreading).

We observe no systematic dependency on $|Q|$ for $\textsc{Seq}_{\textsf{Alias}}$ rendering it a good choice for very large state spaces. While it is up to a factor of 2.0 slower compared to $\textsc{Seq}^{\textsf{prefetch}}_{\textsf{Array}}$, the algorithm might be preferable in a parallel setting.

Memory Footprint and Parallelism. Due to the stochastic nature of the protocols, we expect that in almost all applications several runs of the same protocol are required to derive statistically significant results. On modern machines with many processor cores, one should be able to maximize the throughput by executing multiple independent simulations in parallel. As visualized in Figure 6, most simulators scale well with the number of threads and typically achieve a self-speedup of 40 to 50 times using 40 cores (plus HyperThreading) at $n = 2^{30}$. A notable exception is $\textsc{Seq}^{\textsf{prefetch}}_{\textsf{Array}}$, which reaches only a speedup of 30 as it saturates the memory controllers of both CPU sockets.

Another aspect of parallel execution is the memory footprint. Since $\textsc{Seq}_{\textsf{Array}}$ requires constant memory per agent, our implementations of $\textsc{Seq}_{\textsf{Array}}$ and $\textsc{Seq}^{\textsf{prefetch}}_{\textsf{Array}}$ allocate in excess of 320 GB main memory to execute 80 processes in parallel. Although, this number can be reduced by constant factors using a more efficient representation of states, it has to be contrasted to the competing algorithms with a state space of only a few kilobytes[5] for the same campaign.

[5] We report no exact numbers as the system's process overheads exceed the simulators' internal states.

8 Conclusions and Open Problems

We considered the simulation of large population protocols to allow the experimental investigation of slowly scaling observables in such systems. Two algorithm classes are discussed.

Sequential simulators carry out each interaction one after the other. We analyze the variants $\text{SEQ}_{\mathsf{Array}}$, $\text{SEQ}_{\mathsf{Linear}}$, $\text{SEQ}_{\mathsf{BST}}$, and $\text{SEQ}_{\mathsf{Alias}}$ which differ in the data structures maintaining the agents' states, and demonstrate substantial differences in their practical performances. As a by-product, we describe the Dynamic Alias Table which might be independently applicable.

Batched simulators coalesce interactions to achieve asymptotical speed-ups for protocols with a limited number of states. Our implementation then simulates more than 2^{50} interactions in 400 s which is several orders of magnitudes larger than the fastest sequential simulator.

Possible Extensions. In some variants of the population model it is assumed that interactions are limited to some communication network. Since we store the configurations in our batched simulators as a multiset, it is not clear how to directly adapt our approach. We believe this might be an interesting extension of our work.

Further variants are concerned with the way how agents interact. It is straightforward to adapt our simulator software to a setting where, e.g., a random matching of agents interacts in each time step. Furthermore, our approach could be generalized to a setting where more than two agents interact. In this case we are in need of good heuristics for partitioning the transition-tensor, since the work for updating a batch grows exponentially in the number of interacting agents. In general, sampling the interaction counters is a frequent and costly task. Therefore, any improved heuristic to sample from the $|Q| \times |Q|$ matrix using only $o(|Q|^2)$ variates would yield a measureable benefit for the total runtime of a simulation.

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In *SODA*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.

2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *SODA*. SIAM, 2018. doi:10.1137/1.9781611975031.144.

3 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *ICALP (2)*, volume 9135 of *LNCS*, pages 479–491. Springer, 2015. doi:10.1007/978-3-662-47666-6_38.

4 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In *PODC*, pages 47–56. ACM, 2015. doi:10.1145/2767386.2767429.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.

6 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC*, pages 292–299. ACM, 2006. doi:10.1145/1146381.1146425.

7 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008. doi:10.1007/s00446-008-0067-z.

8 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Comput.*, 21(2):87–102, 2008. doi:10.1007/s00446-008-0059-z.

9 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/s00446-007-0040-2.

10 Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. *Parallel Process. Lett.*, 1:11–18, 1991.

11 Richard Arratia, Skip Garibaldi, and Joe Kilian. Asymptotic distribution for the birthday problem with multiple coincidences, via an embedding of the collision process. *Random Struct. Algorithms*, 48(3):480–502, 2016. doi:10.1002/rsa.20591.

12 James Aspnes and Eric Ruppert. An introduction to population protocols. *Bull. EATCS*, 93:98–117, 2007.

13 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $O(\log^{3/2} n)$ parallel time population protocol for majority with $O(\log n)$ states. In *PODC*, page to appear, 2020.

14 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Majority & stabilization in population protocols. *CoRR*, abs/1805.04586, 2018. arXiv:1805.04586.

15 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with o(log5/3 n) stabilization time and theta(log n) states. In *DISC*, volume 121 of *LIPIcs*, pages 10:1–10:18. Schloss Dagstuhl, 2018. doi:10.4230/LIPIcs.DISC.2018.10.

16 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *STOC*, pages 119–129, 2020.

17 Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. *CoRR*, 2020. arXiv:2005.03584.

18 Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and efficient leader election. In *SOSA@SODA*, volume 61 of *OASICS*, pages 9:1–9:11. Schloss Dagstuhl, 2018. doi:10.4230/OASIcs.SOSA.2018.9.

19 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Brief announcement: Population protocols for leader election and exact majority with $O(\log^2 n)$ states and $O(\log^2 n)$ convergence time. In *PODC*. ACM, 2017. doi:10.1145/3087801.3087858.

20 Paul Bratley, Bennett L. Fox, and Linus Schrage. *A guide to simulation, 2nd Edition*. Springer, 1987.

21 Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2:656, 2012. doi:10.1038/srep00656.

22 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755, 2013. doi:10.1038/nnano.2013.189.

23 Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. doi:10.1007/978-1-4613-8643-8.

24 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. In *DISC*, volume 9363 of *LNCS*, pages 602–616. Springer, 2015. doi:10.1007/978-3-662-48653-5_40.

25 Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. *SIAM J. Control and Optimization*, 50(3):1087–1109, 2012. doi:10.1137/110823018.

26 Robert Elsässer and Tomasz Radzik. Recent results in population protocols for exact majority and leader election. *Bull. EATCS*, 126, 2018. URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546.

27 Bennett L. Fox. Generating markov-chain transitions quickly: I. *INFORMS J. Comput.*, 2(2):126–135, 1990.

28 Leszek Gasieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *SODA*. SIAM, 2018. doi:10.1137/1.9781611975031.169.

29 Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *SPAA*, pages 93–102. ACM, 2019. doi:10.1145/3323165.3323178.

30 Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. Maintaining discrete probability distributions optimally. In *ICALP*, volume 700 of *LNCS*, pages 253–264. Springer, 1993.

31 Lorenz Hübschle-Schneider and Peter Sanders. Parallel weighted random sampling. In *ESA*, volume 144 of *LIPIcs*, pages 59:1–59:24. Schloss Dagstuhl, 2019.

32 Fabian Kuhn and René Struik. Random walks revisited: Extensions of pollard's rho algorithm for computing multiple discrete logarithms. In *Selected Areas in Cryptography*, volume 2259 of *LNCS*, pages 212–229. Springer, 2001. doi:10.1007/3-540-45537-X_17.

33 Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. In *SODA*, pages 361–370. ACM/SIAM, 1993.

34 George B. Mertzios, Sotiris E. Nikoletseas, Christoforos L. Raptopoulos, and Paul G. Spirakis. Determining majority in networks with local interactions and very small local memory. In *ICALP (1)*, volume 8572 of *LNCS*, pages 871–882. Springer, 2014. doi:10.1007/978-3-662-43948-7_72.

35 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In *NCA*, pages 35–42. IEEE Computer Society, 2015. doi:10.1109/NCA.2015.35.

36 Sanguthevar Rajasekaran and Keith W. Ross. Fast algorithms for generating discrete random variates with changing distributions. *ACM Trans. Mod. Comp. Sim.*, 3(1), 1993.

37 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *ESA*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004. doi:10.1007/978-3-540-30140-0_69.

38 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Nat. Comput.*, 7(4):615–633, 2008. doi:10.1007/s11047-008-9067-y.

39 Ernst Stadlober. Ratio of uniforms as a convenient method for sampling from classical discrete distributions. In *Winter Simulation Conference*, pages 484–489. ACM Press, 1989. doi:10.1145/76738.76801.

40 Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Eng.*, 17(9):972–975, 1991. doi:10.1109/32.92917.

41 Alastair J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Math. Soft.*, 3(3), 1977. doi:10.1145/355744.355749.