# Names Trump Malice: Tiny Mobile Agents Can Tolerate Byzantine Failures

Rachid Guerraoui[1] and Eric Ruppert[2]

[1] EPFL
[2] York University

**Abstract.** We introduce a new theoretical model of ad hoc mobile computing in which agents have severely restricted memory, highly unpredictable movement and no initial knowledge of the system. Each agent's memory can store $O(1)$ bits, plus a unique identifier, and $O(1)$ other agents' identifiers. Inputs are distributed across $n$ agents, and all agents must converge to the correct output value. We give a universal construction that proves the model is surprisingly strong: It can solve any decision problem in $NSPACE(n \log n)$. Moreover, the construction is robust with respect to Byzantine failures of a constant number of agents.

## 1 Introduction

Technological advances have made it possible to build ad hoc networks of mobile tiny computing devices that can perform control and monitoring applications without any specific infrastructure. Much work has been devoted to designing and experimenting on such networks, but little has focussed on devising theoretical models to capture their inherent power and limitations. Without such a model, it is impossible to verify the correctness of an algorithm or determine whether some performance bottleneck results from an inherent complexity lower bound or is merely an artifact of a particular environment. Ideally, a theoretical model should be weak enough to cover a large spectrum of environments, yet strong enough to capture what can indeed be computed in such systems.

An elegant candidate for a theoretical framework is the *population protocol* model of Angluin *et al.* [1], which makes absolutely minimal assumptions. It assumes totally asynchronous agents, no system infrastructure, $O(1)$ bits of memory per agent and makes no assumptions about agents' mobility patterns except for a fairness condition, which ensures agents are not forever disconnected. To study computability in the model, they assumed inputs are distributed across all agents and, after a sequence of pairwise interactions, each agent must converge to the correct output. Angluin *et al.* [3] proved the model can solve exactly those decision problems expressible by first-order formulas in Presburger arithmetic.

Unfortunately, this class of solvable problems is fairly small. For instance, it does not include multiplication. Moreover, even for this restricted class, algorithms tolerated no failures or, at worst, a fixed number of benign failures [6]. However, in the hostile environments where ad hoc mobile networks are deployed, arbitrary failures of some agents are often expected. Even one such failure can

prevent any non-trivial computation by population protocols. (See Sect. 3.)

A key obstacle to more sophisticated computations and fault-tolerance in the population protocol model is anonymity: all agents behave identically. Although a tiny device's memory is often very constrained, it is usually sufficient to store a unique identity. Typically, a modern tiny device consists of a micro-controller such as the MSP430 [11], which has 16 KB of RAM, possibly with a flash-memory component, such as the M25PX64 [9], which stores a few additional megabytes that are more expensive to access. Devices are often equipped with a unique identifier. For example, they might contain Maxim's DS2411 chip, which stores just 64 bits of ROM and is set by the factory to store a unique serial number.

Our goal is to define a minimal model of constrained, yet uniquely identified, tiny devices. Our model resembles population protocols, but we assume all $n$ agents have unique identifiers (ids) and can store $O(1)$ other agents' ids. We call our model the *community protocol* model, thinking of a community as a collection of unique individuals, in contrast to a population, which is merely an agglomeration of a nameless multitude. We assume ids are stored in ROM (as in the DS2411 chip), so that Byzantine agents cannot alter their ids. We restrict the usage of ids to their fundamental purpose, *identification*, by assuming algorithms can only compare ids. (An algorithm cannot, for example, perform arithmetic on ids.) In addition to *having* ids, the ability of agents to *remember* other ids is crucial as, otherwise, the model would be as weak as population protocols. As in the population protocol model, a single algorithm works for all values of $n$.

Our main result describes how a community protocol can simulate a Turing machine. Thus, $n$ agents, whose collective memory capacity stores $O(n \log n)$ bits, can solve any decision problem that is in $NSPACE(n \log n)$, even though that memory is scattered across agents with unpredictable movement. (We focus on decision problems, but the results easily generalize to computing functions.) Furthermore, our simulation is resilient to a constant number of Byzantine failures. So, although community protocols have only slightly more memory than population protocols, they are much more powerful: they solve a much wider class of problems and tolerate Byzantine failures.

There are several challenges to overcome in designing the simulation. The system's initially unstructured memory must be organized to allow tasks like garbage collection. Similarly, the agents must organize themselves for a division of labour. Furthermore, both of these tasks must be resilient to Byzantine failures. Another key difficulty is the impossibility of determining when these initial set-up tasks are complete, so it must be possible to restart any tasks that depend on them (while ensuring that Byzantine agents cannot cause spurious restarts).

We use the condition-based approach [8] to characterize the power of community protocols. A Byzantine agent can immediately change or discard its portion of the input. Thus, there must be preconditions that ensure such changes cannot affect the outcome. For example, to compute the majority value of input bits, there must be a precondition that the difference between the number of 0's and the number of 1's is at least $2f$; otherwise $f$ Byzantine agents flipping their input bit would cause any algorithm to produce an incorrect output.

**Theorem 1.** *Any decision problem in $NSPACE(n \log n)$ can be computed by a community protocol in a way that tolerates $f = O(1)$ Byzantine agents, given preconditions ensuring the output does not change if up to $f$ of the input characters are changed and up to $3f + 1$ of the input characters are deleted.*

This leaves a slight gap: clearly, changes to $f$ inputs must not affect the output, but our construction requires slightly stronger preconditions. The proof of Theorem 1 generalizes: if each agent could store more ids, they could tolerate more failures: if each agent can store $O(f^2)$ ids, then any problem in $NSPACE(fn \log n)$ can be solved tolerating $f$ Byzantine failures (again, with preconditions). In contrast, most work on Byzantine failures focusses on the number of shared objects or messages, not the amount of local memory used.

## 2 Computation Models

After a brief, informal description of population protocols, we define community protocols. We also describe a version of pointer machines used in our proof.

In the **population protocol** model [1], a system is a collection of agents. Each agent can be in one of a finite number of possible states. Each agent has an input value, which determines its initial state. When two agents meet, they exchange information about their states and simultaneously update their own states, according to a joint transition function. Each possible agent state has an associated output value. The input assignment, transition function and the association of an output value with each state provides a complete specification of an algorithm. Algorithms are *uniform*: they do not depend on the size of the system. An execution begins with a collection of agents assigned their initial values and proceeds by an infinite series of pairwise interactions. An execution is *fair* if every system configuration that is forever reachable is eventually reached.

For a finite alphabet $A$, $A^*$ denotes the set of finite strings over $A$, and $A^+$ denotes the set of non-empty finite strings over $A$. Let $\Sigma$ be the finite set of input values for individual agents. Let $Y$ be the set of possible outputs. An algorithm (stably) *computes* a function $f : \Sigma^+ \to Y$ if, for all $x \in \Sigma^+$, every fair execution starting with the characters of $x$ assigned as inputs to $|x|$ agents eventually stabilizes to output $f(x)$ (*i.e.*, from some time onward, all agents output $f(x)$).

We extend the population protocol model to obtain our **community protocol** model by assigning unique ids to agents. Let $U$ be an infinite ordered set containing all possible ids. A community protocol *algorithm* is specified by: (1) a finite set, $B$, of possible basic states, (2) an integer $d \geq 0$ representing the number of ids that can be remembered by an agent, (3) an input map $\iota : \Sigma \to B$, (4) an output map $\omega : B \to Y$, and (5) a transition relation $\delta \subseteq Q^4$, where $Q = B \times (U \cup \{\perp\})^d$. The state of an agent stores an element of $B$, together with up to $d$ ids. The first of the $d$ slots will be initialized with an agent's unique id. If any of the $d$ slots is not currently storing an id, it contains a null id $\perp \notin U$. Thus, $Q$ is the set of possible agent states. The transition relation indicates what happens when two agents interact: if $(q_1, q_2, q_1', q_2') \in \delta$, it means that when two agents in states $q_1$ and $q_2$ meet, they can move into states $q_1'$ and $q_2'$, respectively.

As in population protocols, algorithms are uniform: they cannot use any bound on the number of agents, so $U$ is infinite. Thus, the $d$ slots in an agent's state intended for ids could store arbitrary amounts of information. To avoid this, we require that agents store only ids they have learned from other agents. From a practical perspective, this implies that if the algorithm is actually implemented in a real system, $O(\log n)$ bits of memory per agent will suffice. This is analogous to the common assumption in models like random access machines that the number of bits that fits into one memory word is logarithmic in the size of the problem. To keep the model minimal, we require algorithms' operations on ids to be *comparison-based*. These two constraints are formalized as follows.
(1) If $(q_1, q_2, q_1', q_2') \in \delta$ and $id$ appears in $q_1'$ or $q_2'$ then $id$ appears in $q_1$ or $q_2$.
(2) Consider a transition $(q_1, q_2, q_1', q_2') \in \delta$. Let $u_1 < u_2 < \cdots < u_k$ be all ids that appear in any of the four states $q_1, q_2, q_1'$ and $q_2'$. Let $v_1 < v_2 < \cdots < v_k$ be ids. If $\rho(q)$ is the state obtained from $q$ by replacing all occurrences of each id $u_i$ by $v_i$, then we require that $(\rho(q_1), \rho(q_2), \rho(q_1'), \rho(q_2'))$ is also in $\delta$.

A *configuration* of the algorithm consists of a finite vector of elements from $Q$. Let $X \subseteq \Sigma^+$ be the set of all possible input strings. An *initial configuration* for $n$ agents is a vector in $Q^n$ of the form $(\langle \iota(x_j), u_j, \perp, \perp, \ldots, \perp \rangle)_{j=1}^n$ where $u_1, \ldots, u_n$ are distinct elements of $U$ and $x_j$ is the input to the $j$th agent. The input string represented by this initial configuration is $x = x_{\pi(1)} x_{\pi(2)} \ldots x_{\pi(n)} \in X$, where $\pi$ is the permutation of $\{1, \ldots, n\}$ such that $u_{\pi(1)} < u_{\pi(2)} < \cdots < u_{\pi(n)}$. (In other words, the input string $x$ is the string of input symbols ordered by agent ids.)

We first define fair executions for the failure-free case. If $C = (q_1, \ldots, q_n)$ and $C' = (q_1', \ldots, q_n')$ are two configurations, we say $C$ *reaches* $C'$ *in a single step* (denoted $C \to C'$) if there are indices $i \neq j$ such that $(q_i, q_j, q_i', q_j') \in \delta$ and for all $k$ different from $i$ and $j$, $q_k = q_k'$. A failure-free *execution* on input string $x \in X$ is an infinite sequence of configurations $C_0, C_1, \ldots$, such that $C_0$ is an initial configuration for $x$ and $C_i \to C_{i+1}$ for all $i \geq 0$. In reality, several pairs of agents may interact simultaneously, but simultaneous interactions can simply be listed, in any order, in the execution since they are independent of one another. A failure-free execution is called *fair* if, for each $C$ that appears infinitely often and for each $C'$ such that $C \to C'$, $C'$ also appears infinitely often.

We now define executions in the presence of Byzantine agents. Since the order of agents within configuration vectors is unimportant (and invisible to the algorithm itself), assume for convenience that the Byzantine agents occupy the last components of the vectors. An execution with $t \leq n$ Byzantine failures is an infinite sequence of configurations $C_0, C_1, \ldots$ such that $C_1$ is an initial configuration of the algorithm, and for all $i \geq 0$, either (1) $C_i \to C_{i+1}$ or (2) the first $n - t$ components are unchanged between $C_i$ and $C_{i+1}$ and the ids of the last $t$ components are unchanged. Case (2) allows Byzantine agents to undergo arbitrary state changes (without altering their ids). In particular, they can change the ids in the other $d - 1$ slots of their states to any elements of $U$.

Byzantine agents should be exempted from fairness requirements. In particular, a Byzantine agent might never interact with any agent. For simplicity assume that, in any execution, each Byzantine agent can only have a finite (but

unbounded) number of different states. Thus, the entire system only has a finite (but unbounded) number of different configurations in any execution. Consider an execution $C_1, C_2, \ldots$ with $t$ Byzantine agents. Let $D_i$ be the first $n - t$ components of $C_i$. The execution is *fair* if, whenever infinitely many of the $D_i$'s are equal to $D$ and $D \to D'$, then infinitely many of the $D_i$'s are equal to $D'$.

An algorithm *computes* a function $f : \Sigma^+ \to Y$ tolerating $f$ Byzantine failures if, for all input strings $x \in X$, every fair execution $C_1, C_2, \ldots$ with $t \leq f$ failures starting from any initial configuration $C_1$ for $x$ converges to $f(x)$, *i.e.*, there is an $i$ such that for every $j > i$ and every state $(b, u_1, \ldots, u_d)$ of a correct agent in $C_j$, $\omega(b) = f(x)$. (A population protocol is a special case with $d = 0$.)

Instead of simulating a Turing machine directly, our community protocol simulates a pointer machine, which has a memory structured more like a community protocol's. Several types of pointer machines were developed independently. (See [5] for a survey.) Here, we describe the slightly revised version of Schönhage's **storage modification machine** (SMM) model [10] used in our construction. An SMM represents a single computer, not a distributed system. Its memory stores a finite directed graph of constant out-degree, with a distinguished node called the *centre*. The edges of the graph are called *pointers*. The edges out of each node are labelled by distinct *directions* drawn from a finite set $\Delta$. Any string $x \in \Delta^*$ can be used as a reference to the node (denoted $p^*(x)$) that is reached from the centre by following the sequence of pointers labelled by the characters of $x$. The basic operations of an SMM allow the machine to create nodes, change pointers and follow paths of pointers. Formally, an SMM is specified by a finite input alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$, the set $\Delta$, and a programme, which is a finite list of consecutively numbered instructions. Inputs to the SMM are finite strings from $\Sigma^*$. Programmes use instructions of the following types.

- **new** creates a new node, makes it the centre, and sets all its outgoing pointers to the old centre.
- **recentre** $x$, where $x \in \Delta^+$, changes the centre of the graph to $p^*(x)$.
- **set** $x\delta$ **to** $y$, where $x, y \in \Delta^*$ and $\delta \in \Delta$, changes the pointer of node $p^*(x)$ that is labelled by $\delta$ to point to node $p^*(y)$.
- **if** $x = y$ **then goto** $\ell$, where $x, y \in \Delta^*$, jumps to line $\ell$ if $p^*(x) = p^*(y)$.
- **input** $\ell_1, \ell_2, \ldots, \ell_r$, where $\ell_1, \ldots, \ell_r$ are line numbers, consumes the next input character (if there is one) and jumps to line $\ell_i$ if that character is $\sigma_i$.
- **output** $o$, where $o \in \{0, 1\}$, causes the machine to halt and output $o$.

When a node becomes unreachable from the centre, it can be dropped from the graph, since it plays no further role in the computation. Space complexity is measured by the maximum number of (reachable) nodes present at any one time during the execution. Our instruction set differs slightly from Schönhage's. We have omitted instructions that can be trivially implemented from ours and input/output instructions for online computation. We have given a separate name to the **recentre** instruction, because it is handled separately in our construction.

As shown by van Emde Boas [12], an SMM can simulate a Turing machine.

**Theorem 2.** *Any language decided by a Turing machine using $O(S \log S)$ space can be decided by an SMM using $S$ nodes* [12].

We introduce a *nondeterministic SMM* (NSMM), adding choose instructions:

• **choose** $\ell_0, \ell_1$, where $\ell_0$ and $\ell_1$ are line numbers, causes the machine to transfer control either to line $\ell_0$ or to line $\ell_1$ nondeterministically.

We define acceptance of a string by an NSMM as for nondeterministic Turing machines: an NSMM accepts a string $x$ if and only if *some* execution on input $x$ outputs 1. The following theorem can be proved in exactly the same way as Theorem 2, so it will suffice to simulate an NSMM using community protocols.

**Theorem 3.** *Any language decided by a nondeterministic Turing machine using $O(S \log S)$ space can be decided by an NSMM using $S$ nodes.*

## 3 Population Protocols Cannot Handle Byzantine Agents

A function $f : X \to Y$ is called *trivial* if the output can be determined from any single input character, *i.e.,* for any strings $x, y \in X$ that both contain a common character, $f(x) = f(y)$. Clearly, population protocols can compute any trivial function, even with Byzantine failures. We prove the converse also holds.

**Theorem 4.** *Any function computable by a population protocol tolerating one Byzantine agent is trivial.*

*Proof.* Consider an algorithm that computes $f$, tolerating one Byzantine failure. Let $x$ and $y$ be two strings with a common character $a$. We prove $f(x) = f(y)$. Let $n = |x| + |y| - 1$ and $p_1, p_2, \ldots, p_n$ be agents. Let $E$ be a fair, failure-free execution where $p_1, \ldots, p_{|x|}$ have the characters of $x$ as inputs and $p_{|x|+1}, \ldots, p_n$ have the characters of $y$, except for one copy of $a$, as inputs. Furthermore, assume $p_1$ has input $a$ in $E$. (The input vector of $E$ may or may not be in $X$.)

Let $E'$ be the execution of $|x|$ agents $p_1, \ldots, p_{|x|}$ with input string $x$, where all agents except $p_{|x|}$ behave as in $E$, and $p_{|x|}$ behaves in a Byzantine manner, simulating the actions of all of the agents $p_{|x|}, \ldots, p_n$ in $E$. The fairness of $E'$ follows from the fairness of $E$. Thus, in $E'$, $p_1$ must stabilize to the output $f(x)$. Since $p_1$ cannot distinguish $E$ from $E'$, $p_1$ must stabilize to $f(x)$ in $E$ too.

A symmetric argument (comparing $E$ to an execution where $p_1, p_{|x|+1}, \ldots, p_n$ have input $y$ and $p_{|x|+1}$ is Byzantine, simulating the actions of $p_2, \ldots, p_{|x|+1}$) shows $p_1$ stabilizes to $f(y)$ in $E$. Thus, $f(x) = f(y)$. □

## 4 Simulating an NSMM with a Community Protocol

We now outline the proof of Theorem 1, starting with a high-level overview. By Theorem 3, it suffices to design a community protocol that simulates a NSMM that uses $O(n)$ nodes. Due to space restrictions, we omit detailed proofs.

First, the agents organize themselves into a virtual ring. They then divide up into $O(f)$ clusters of approximately equal size using the ring structure. Each cluster collectively undertakes a simulation of the NSMM on the entire input string. In the simulation, each agent stores $O(f)$ nodes of the NSMM's graph data

structure. Each agent can determine, by itself, whether another agent belongs to its own cluster, preventing other clusters' Byzantine agents from interfering with its cluster's simulation. Thus, a majority of clusters will simulate the NSMM correctly. Each agent takes the majority output value of all clusters as its output.

Agents cannot know when the ring structure has stabilized, since a previously inert agent can begin interacting at any time. Thus, agents cannot wait until the structure has stabilized before starting to simulate the NSMM. Instead, they begin the simulations right away. Then, whenever the ring structure changes, the simulations are restarted. (Our algorithm ensures that Byzantine agents cannot cause spurious restarts.) We prove that simulations that begin after the ring structure has stabilized eventually converge to the correct answer.

The mechanism for restarting simulations is also used to handle the non-determinism of the NSMM. The system must output 1 if *some* execution of the NSMM outputs 1. Whenever the cluster's simulation outputs 0, it restarts, continuing to search for an execution that outputs 1. Some executions of the NSMM may run forever. To handle this, the simulation can make a non-deterministic decision to restart the simulation at any time. Fairness ensures that, if some execution of the NSMM outputs 1, correct clusters will eventually find it.

**Building the Ring Structure**. All agents can be imagined as forming an abstract ring, where the agents are ordered (clockwise) by their ids. The successor of the agent with the largest id is the agent with the smallest id. Our protocol makes use of this ring in several ways. In particular, an agent will sometimes have to to traverse the ring, meeting (almost) all the agents in order.

Initially, agents have no information about the abstract ring, so they must learn about it. Without failures, it would suffice for each agent to learn its successor's id. However, a Byzantine agent could disrupt traversals of the ring by lying about its successor or refusing to meet an agent traversing the ring. To avoid these problems, we build some redundancy into the agents' knowledge of the ring. Let $s = 8f + 4$. Each agent $p$ has a *successors* field which stores the ids of $p$'s $s$ closest successors in the abstract ring that $p$ has learned of so far.

An agent can learn about its successors by meeting them directly. If $q$ is correct, it eventually meets all of its predecessors who can record $q$'s id in their *successors* fields. However, if $q$ is Byzantine, $q$ may meet some of its predecessors, but not others. To facilitate orderly ring traversals, we ensure $q$'s id eventually either appears in the *successors* fields of many of its predecessors (in which case no traversal skips $q$) or very few of them (in which case every traversal skips $q$).

To achieve this agreement about the ring structure, agents gossip about their successors. If $p$ sees a new id $x$ in another agent's *successors* field, it can non-deterministically choose to begin gathering evidence that $x$ is a real id: it then remembers ids of agents that have $x$ in their *successors* fields. If $p$ meets $f + 1$ such agents, it concludes that at least one correct agent believes $x$ is the id of a real agent, and can add $x$ to its own *successors* field. The following lemma is an easy consequence of the fact that Byzantine agents cannot forge their own ids.

**Lemma 1.** *A correct agent's successors fields always contains only ids of actual agents in the system and its successors field eventually stabilizes.*

The next lemma describes the degree of agreement achieved by gossiping when some *successors* fields have stabilized. We use $furthest(p)$ to denote the last id of $p.successors$ (taken in clockwise order around the ring, starting from $p$).

**Lemma 2.** *Let $\mathcal{C}$ be a set of correct agents whose successors fields have stabilized at time $T$. For some $\mathcal{B} \subseteq \overline{\mathcal{C}}$ the following properties always hold after $T$.*
*(1) For each correct agent $p$, after $p.successors$ has stabilized, it contains the ids of all agents of $\mathcal{C} \cup \mathcal{B}$ that are located after $p$ and before the agent with id $furthest(p)$ in the abstract ring.*
*(2) For each agent $q \notin \mathcal{C} \cup \mathcal{B}$, at most $f$ of the agents in $\mathcal{C}$ have the id of $q$ in their successors fields.*

**Traversing the Ring**. Our construction will require an agent $p$ to traverse the ring structure, performing some action upon each agent it meets. Ideally, $p$'s traversal should visit every correct agent. However, $p$ cannot wait forever to meet a particular agent, since that agent could be Byzantine. Thus, we are satisfied if $p$ performs its action on almost all other agents. To do the traversal, $p$ maintains a sorted list of $s$ ids from one segment of the ring in an array field called $current[1..s]$. To advance the traversal, the first element of *current* is removed, all others are shifted left and a new id is added in $current[s]$. To begin a traversal, $p.current$ is initialized to $p.successors$. Agent $p$ then waits until it meets at least $s-f$ of those $s$ agents, remembering each of *their successors* fields in its own state. (This is the most memory-intensive part of the construction: $p$ must remember $\Theta(f^2)$ ids.) Agent $p$ then adds to $p.current$ the next id in the ring (beyond $p.current[s]$) that appears in at least $3f+2$ of those $s-f$ agents' *successors* fields. We show that this makes it impossible for the $f$ Byzantine agents to misdirect the traversal. The traversal proceeds by iterating this process.

Naturally, traversals will not work properly before the ring structure has stabilized. However, the traversals do work correctly once *most successors* fields have stabilized. To make this more precise, consider any execution. Let $T$ be the latest time such that changes are made to the *successors* fields of exactly $f+1$ correct agents after $T$. (This time exists, by Lemma 1.) Let $\mathcal{C}$ be the set of correct agents whose *successors* fields do not change after $T$. By definition of $T$, $|\mathcal{C}| \geq n-2f-1$. Define $\mathcal{B}$ to satisfy Lemma 2 for these choices of $T$ and $\mathcal{C}$. The following lemma, which relies heavily on Lemma 2's guarantee that there is eventually a high degree of agreement among agents' *successors* fields, states that a traversal's advances are orderly if the traversal is started after $T$.

**Lemma 3.** *Suppose a correct agent $p$ begins a traversal after $T$ and after its own successors field has stabilized. Then, at all times, $p.current$ contains all agents of $\mathcal{C} \cup \mathcal{B}$ that lie within a segment of the abstract ring (sorted in clockwise order). Furthermore, each advance will insert into $p.current[s]$ the id of the next agent of $\mathcal{C} \cup \mathcal{B}$ that is after (in clockwise order) the id previously stored in $p.current[s]$.*

Before advancing a traversal, $p$ must meet $s-f$ of the agents whose ids are in $p.current$. Since the traversal is intended to accomplish some task, there may be an additional *external condition*, defined by the algorithm, which must be

satisfied before the traversal can advance. The following lemma says that if that external condition is eventually satisfied, the traversal will indeed make progress.

**Lemma 4.** *If a correct agent $p$ begins a traversal after $T$ and after $p.successors$ has stabilized, then, in any suffix of the execution, either the external condition for advancing is unsatisfied infinitely often, or an advance eventually occurs.*

**Global Resets**. When the ring structure changes, agents recompute their clusters and re-start all simulated executions. Each time an agent $p$ changes its *successors* field, it initiates a *global reset*, which tells (almost) every agent in the system to perform a *local reset* of its own memory. Local resets are described below. Agent $p$ does this by traversing the ring, giving a *reset request* to each agent it visits. To ensure Byzantine agents cannot cause spurious local resets, an agent performs a local reset only after receiving reset requests from $f+1$ different agents, not all of which can be Byzantine. Before the ring structure stabilizes, a global reset has unpredictable behaviour. However, Lemma 1 ensures correct agents eventually stop initiating global resets, and this implies the next lemma.

**Lemma 5.** *Each correct agent performs a finite number of local resets.*

By Lemma 3, global resets give reset requests to most agents after $T$. Thus, most agents receive $f + 1$ reset requests after $T$ and do a local reset:

**Lemma 6.** *Each correct agent in $\mathcal{C} \cup \mathcal{B}$ performs a local reset after $T$.*

**Local Resets and Clustering**. Local resets occur because the ring structure has changed. Agents then recompute the clusters and restart the simulations of the NSMM. Clustering (eventually) divides agents into $g = 4f + 3$ clusters of roughly the same size by chopping the ring into $g$ sections. When an agent performs a local reset, it first recomputes the cluster boundaries. If the agent is the leader of its cluster (*i.e.*, the agent with the lowest id in the cluster), then it restarts the simulation of the NSMM within its cluster. When a non-leader does a local reset, it also asks the leader of its cluster to restart the simulation.

We first describe how an agent computes the boundaries of all clusters after its local reset. It initiates *two* traversals, called the *slow traversal* and the *swift traversal*. The slow traversal advances once every $g + 1$ advances of the swift traversal. Whenever the swift traversal overtakes the slow traversal, the agent increments a counter and remembers the id of the agent where this crossing occurred as a cluster boundary. (As an analogy, the locations on a clock face where the minute hand passes the hour hand divide the face into 11 equal segments.)

By Lemma 6, all agents in $\mathcal{C}$ recompute clusters after $T$. Each agent remembers the cluster boundaries by storing the lowest id in each cluster. By Lemma 3, they will perform their slow and swift traversals identically in this final clustering, and therefore all compute identical cluster boundaries. In this final clustering after $T$, a cluster is called *correct* if it contains only agents in $\mathcal{C}$. We prove that correct clusters simulate the NSMM correctly after stabilization. A careful computation of exactly where the swift traversal passes the slow traversal proves these clusters are roughly equal in size.

**Lemma 7.** *Each cluster includes at least* $\frac{n-2f-1}{g} - 2$ *agents.*

**Simulating an NSMM within a Cluster**. A cluster's simulation of the NSMM is driven by its leader, $p$. Other agents in the cluster store the NSMM's graph data structure, which contains up to $cn$ nodes for some constant $c$. Each agent stores up to $2cg$ nodes, ensuring that the collective capacity of the agents in the cluster is big enough to store $cn$ nodes, by Lemma 7. A pointer to a node is represented by an id-index pair: the id identifies the agent storing the node and the index identifies one of the nodes stored by that agent. Agent $p$ stores in its state a programme counter that describes what line of the NSMM programme it is simulating and keeps an id-index pair that locates the graph's centre. It simulates each instruction of the NSMM's programme one by one as follows.

To handle **input** instructions, $p$ traverses the ring, starting from the agent with the lowest id. When the simulation reaches an **input** instruction, $p$ advances the traversal and consumes the input of one agent. Ideally, the input symbols of all agents would be consumed. However, a traversal visits only those agents in $\mathcal{C} \cup \mathcal{B}$, so the simulation will miss inputs from any agents outside that set. Furthermore, any Byzantine agents in $\mathcal{B}$ might refuse to cooperate with the traversal, so $p$'s input traversal must skip their inputs. Unfortunately, $p$ does not know whether traversed agents belong to $\mathcal{C}$ or $\mathcal{B}$. So $p$ guesses which agents to skip during the traversal, never skipping more than $f$ of them. If $p$ guesses incorrectly, it might choose *not* to skip an agent that is, in fact, Byzantine, and the Byzantine agent may refuse to meet $p$ and provide an input character to the simulation. In this case, $p$ gives up and restarts the simulation from scratch. Fairness guarantees that, if an accepting execution of the NSMM exists, the simulation will eventually find it when it correctly guesses which agents to skip. The overall effect of this input scheme is that the simulation might miss inputs from up to $3f+1$ agents (because there may be up to $2f+1$ agents outside $\mathcal{C} \cup \mathcal{B}$ and the simulation will skip up to $f$ agents in $\mathcal{C} \cup \mathcal{B}$) and, among the agents from which inputs *are* received, up to $f$ may be altered by Byzantine agents.

To simulate a **recentre** $x$ instruction, $p$ follows the pointers described by the string $x$ one by one. Along the way, $p$ stores the id-index pair locating each node visited. It advances along the path by waiting to meet the agent with that id, and then copies the new id-index pair from that agent's state. When $p$ has finished traversing the path, it updates its *centre* field to point to the new centre.

To simulate a **set** $x\delta$ **to** $y$ instruction, $p$ first follows the pointers represented by the string $x$ to find the agent $q$ storing the node whose pointer must be updated. It then follows the pointers represented by $y$. Both paths are followed in the way described in the previous paragraph. Then, $p$ changes the pointer stored in $q$ to point to the node that it found at the end of the second path.

To simulate an **if** $x = y$ **then goto** $\ell'$ instruction, $p$ follows both paths of pointers as described above, compares the resulting id-index pairs, and updates its programme counter accordingly.

For a **new** instruction, $p$ locates an agent that stores fewer than $2cg$ nodes that are reachable from the centre. Agent $p$ runs a garbage collection to detect unreachable nodes. First, $p$ iterates across all agents in the cluster, marking

all nodes within their memories as unvisited. Then, $p$ executes a depth-first search (DFS) from the centre of the graph data structure. It then waits until it finds an agent $q$ with a node that was not visited during the DFS and replaces that node by the old centre. There will always be such an agent $q$: the NSMM uses $cn$ nodes in the worst case and there is space within the cluster to store $2cg(\frac{n-2f-1}{g} - 2) = c(2n - 4f - 2g - 2) \geq cn$ nodes. Agent $p$ then creates a new centre node in $q$ and updates its own *centre* field.

If the simulation reaches an **output**(1) instruction, then the simulation has successfully found an accepting execution, and no further simulation is necessary. The cluster leader $p$ sets a field of its memory called *clusterOutput* to 1. (This field is initialized to 0 whenever $p$ restarts a simulation.) If the simulation reaches an **output**(0) instruction, it is treated as a failed attempt to find an execution that leads to an output of 1, and $p$ restarts the simulation of the NSMM, continuing to look for some other execution of the NSMM that outputs 1.

When $p$ is supposed to simulate a **choose** $\ell_0, \ell_1$ instruction, it simply changes its *control* field to represent either the beginning of line $\ell_0$ or the beginning of line $\ell_1$, making the choice nondeterministically.

To abandon simulated executions that never halt, $p$ can non-deterministically choose to restart the simulated execution at any time.

**Producing the Output**. Consider any correct cluster. Let $p$ be its leader. Each agent in the cluster performs a local reset after $T$, by Lemma 6, and will realize that it is a member of the cluster. The cluster's simulation of the NSMM is restarted when the last of these local resets occurs (and *p.clusterOutput* is reset to 0). After this, the simulations of executions by the NSMM within the cluster are correct since the cluster contains no Byzantine agents. Thus, *p.clusterOutput* will eventually be changed to 1 only if there is an execution of the NSMM that outputs 1 for some input string obtained from the actual input by deleting at most 3f+1 input characters and changing $f$ of them (as described above). Conversely, if such an accepting execution of the NSMM exists, the cluster will eventually simulate one, by fairness. Thus, the preconditions imply that *p.clusterOutput* eventually converges to the correct output value.

Whenever an agent meets a cluster leader, it copies that leader's current *clusterOutput* field into its own memory. The output map $\omega$ of the community protocol has each agent output the value that appears in a majority of its own local copies of the clusters' outputs. Since $|\overline{\mathcal{C}}| \leq 2f + 1$, a majority of the $g = 4f + 3$ clusters are correct, so all agents converge to the correct output.

## 5 Conclusion

Many variants of the population protocol model have been studied. See [4] for a survey. Most of the work on population protocols assumes no failures. Delporte-Gallet *et al.* [6] gave a general construction that allows population protocols to tolerate $O(1)$ benign failures (halting failures and transient failures). Here, we used a similar technique of dividing agents into groups, each of which can simulate the entire protocol. However, the mechanisms needed to cope with Byzantine

failures are much more involved. Angluin, Aspnes and Eisenstat [2] described a population protocol that computes majority tolerating $O(\sqrt{n})$ Byzantine failures. However, it is designed for a much more restricted setting, where the scheduler chooses the next interaction randomly and uniformly.

Our model of mobile tiny devices tolerates Byzantine failures, while exploiting the full power of the agents' memory. This work found connections between modern mobile systems, automata theory and Turing machines, using pointer machines as a bridge. Many open questions remain. Can the small gap between the preconditions that are sufficient for our construction and the ones that are necessary for computation be closed? Can $f$ Byzantine failures be tolerated if each agent can store fewer than $\Theta(f^2)$ ids? Because it is so general, our simulation would be extremely slow, but it might be possible to construct much faster simulations with additional assumptions (for example, with a probability distribution on the interactions). Finally, is the ordering on $U$ required to cope with Byzantine failures, or would tests for equality between ids suffice (as in the failure-free case [7])?

**Acknowledgements**. The reviewers provided helpful feedback for this version (and the future full version). This research was funded in part by NSERC.

## References

1. D. Angluin, J. Aspnes, Z. Diamadi, M. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Dist. Comp.*, 18(4):235–253, 2006.
2. D. Angluin, J. Aspnes, and D. Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, July 2008.
3. D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.
4. J. Aspnes and E. Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*. Springer, 2009.
5. A. M. Ben-Amram. Pointer machines and pointer algorithms: an annotated bibliography. Available from `http://www2.mta.ac.il/~amirben`, 1998.
6. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. When birds die: Making population protocols fault-tolerant. In *Proc. 2nd IEEE International Conference on Distributed Computing in Sensor Systems*, pages 51–66, 2006.
7. R. Guerraoui and E. Ruppert. Even small birds are unique: Population protocols with identifiers. Technical Report CSE-2007-04, Dept of Computer Science and Engineering, York University, 2007.
8. A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, Nov. 2003.
9. Numonyx. M25PX64 data sheet. Available from `http://www.numonyx.com/Documents/Datasheets/M25PX64.pdf`, 2007.
10. A. Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, Aug. 1980.
11. Texas Instruments. MSP430 ultra-low-power microcontrollers. Available from `http://focus.ti.com/lit/ml/slab034n/slab034n.pdf`, 2008.
12. P. van Emde Boas. Space measures for storage modification machines. *Inf. Process. Lett.*, 30(2):103–110, Jan. 1989.