Even Small Birds are Unique: Population Protocols with Identifiers

Rachid Guerraoui

Eric Ruppert

Technical Report CSE-2007-04

September 10, 2007

Department of Computer Science and Engineering

4700 Keele Street North York, Ontario M3J 1P3 Canada

Even Small Birds are Unique: Population Protocols with Identifiers

Rachid Guerraoui
EPFL

Eric Ruppert
York University

September 6, 2007

Abstract

Although much research has been devoted to designing and experimenting on *ad hoc* networks of tiny devices, very little has focussed on devising theoretical models to capture the inherent power and limitations of such networks. A notable exception is the population protocol model of Angluin *et al.* [2]. This model is simple and elegant but is sometimes considered too restrictive because of its anonymity: mobile agents have no identities and all look the same.

We investigate in this paper the inherent power of the population protocol model augmented with the ability of each agent to be uniquely identified as well as store a constant number of other agents' identifiers. We provide an exact characterization of what can be computed in this new *community protocol* model: a function can be computed if and only if it is symmetric and in $NSPACE(n \log n)$. This is shown using a simulation of pointer machines.

We also consider the ability of our community protocol model to handle failures. We describe what can be computed when there are a constant number of benign failures and show that non-trivial computations can be achieved even if agents can be Byzantine.

Keywords: population protocols, computability, anonymity, stable computation, mobile computing, Turing machines, pointer machines, fault tolerance, majority.

1 Introduction

God formed . . . every fowl of the air; and brought them unto Adam to see what he would call them: and whatsoever Adam called every living creature, that was the name thereof. (Gen. 2:19)

Angluin *et al.* [2] introduced the population protocol model to describe systems of tiny mobile agents. Each agent is a finite state machine and the movements of the agents are completely unpredictable. When two agents come into range of each other, they can exchange information. As a motivating example, they described a network of monitoring sensors strapped to birds in which two sensors could communicate by radio when their host birds were sufficiently close together. The goal of studying the population protocol model was to see what could be computed with absolutely minimal assumptions about a mobile *ad hoc* network. As such, the model considers totally asynchronous agents, no assumptions about the mobility patterns of the agents (except for a fairness guarantee that ensures agents will not be forever disconnected from the others), only a constant amount of memory per agent, and no system infrastructure. Each agent has an input value and, in every execution, all agents must eventually produce the output that corresponds to the collection of input values.

The population protocol model, along with some variations, has been studied in a series of papers [1, 2, 3, 4, 5, 8, 10]. Angluin *et al.* [2] gave several examples of predicates that can be computed in the population protocol model, and it was later shown that no others are computable [4]. This provided a characterization of what can be computed in the model: it can solve exactly those decision problems expressible by first-order formulas in Presburger arithmetic [16]. (This is essentially first-order arithmetic, using the symbols $+, 0, 1, \wedge, \vee, \neg, \forall, \exists, =, <, (,)$ and variables.)

The class of functions expressible in this way is fairly small. For instance, it does not include multiplication. Even for computing this restricted class of functions, algorithms in the population protocol model assume no failures or, in the worst case, a fixed number of benign failures, say crashes [8]. However, in the kind of hostile environments where *ad hoc* networks of mobile agents are considered appealing, arbitrary failures of a subset of the agents are to be expected, leaving aside the possibility of a dishonest party dropping malicious devices. It is not difficult to see, intuitively, that even a single arbitrary failure can prevent any non-trivial computation in the original population protocol model. (We prove this more precisely in this paper.)

A key obstacle to computing more sophisticated functions and tolerating failures in the population protocol model is its inherent anonymity. There is no notion of agent identity and all agents execute the same algorithm. This lack of identifiers is sometimes seen as too restrictive. Although it is reasonable to assume that the memory of a tiny device is constrained, this memory is usually sufficient to provide each agent with a unique identity.

The motivation of this paper is to relax the anonymity assumption while preserving the constrained nature of tiny devices, following the minimality philosophy of the original population protocol model. In short, we augment the original model by assigning each agent a unique identifier and allowing it to remember a constant number of identifiers of other agents. To avoid ending up with a very powerful model where memory slots intended to store identifiers could encode arbitrarily large amounts of information, thereby circumventing the limited storage capacity, we restrict the usage of identifiers to their fundamental purpose: *identification*. Our restriction indirectly ensures that, when an algorithm is implemented in practice, the size of the slots used to the store identifiers can be bounded, depending on the set of identifiers actually used in the system.

We call the resulting model the *community protocol* model. We use this term, thinking of a community as a collection of unique individuals, as opposed to a population, which is merely an agglomeration of a faceless, nameless multitude. The community protocol model is a strict generalization of the population protocol model that still preserves its spirit. Our model can be viewed as a minimal model for mobile computation by uniquely identified tiny devices. In addition to *having* identifiers, the ability for agents to *remember* some other identifiers is crucial as, otherwise, a unique identity adds no power to the original model. (There would be finitely many different transition functions for a given state set, and thus an infinite number of identifiers that yield identical transition functions. Any computation among those would yield a computation in the original model.)

We show that our community protocol model is significantly stronger than the anonymous population protocol model. We give a precise characterization of what can be computed without failures and give some results about the ability of community protocols to cope with different kinds of failures.

We first prove that, if no failures are assumed, then we can emulate a Turing machine, making very efficient use of the memory space of the system. This result might look surprising at first glance. Consider for instance the simple example of multiplication, which is known to be impossible in the original (anonymous) population protocol model. The problem can be formulated using three input symbols a, b and c, so that the task is computing whether the number of c's is equal to the product of the numbers of a's and b's. The computation could proceed by cancelling one c for every a, b pair. If agents with input a can remember all b's, say in a model without restriction on memory, then the computation is trivial. With our restriction that the a's can only remember a constant number of

b's, the computation is slightly more challenging. Roughly speaking, some structuring of the b's is needed and we will show how to achieve that in the general case of an arbitrary computation, *i.e.*, emulating a Turing machine. We establish a connection between our community protocol model and pointer machines, more specifically non-deterministic storage modification machines. Some of the issues we address in devising the emulation include unpredictable interactions and garbage collection. We prove that a function can be computed in our model if and only if the function is symmetric and in $NSPACE(n \log n)$.

We then consider failures and show that our Turing machine emulation can be extended to tolerate a fixed number of benign failures. The revised characterization is precisely stated using the condition-based approach [15] and its proof applies techniques of Delporte-Gallet *et al.* [8] to make the emulation fault-tolerant.

Finally, we address Byzantine (arbitrary) failures. We show that, in the presence of one Byzantine failure, no meaningful computation can take place in the population protocol model. The impossibility also holds in the community protocol model if a malicious agent can forge identifiers. We prove that if identifiers cannot be forged, then we can perform non-trivial computations (*e.g.*, majority) in the presence of a constant number of arbitrary failures. This is less trivial than with benign failures because a malicious agent can cancel intermediate computations at several agents before betraying itself: the challenge is for honest agents to recognize and remember such agents while using a limited number of identifiers. Characterizing what exactly can be computed in this case is left open.

To summarize, this paper contributes to the exploration of a simple, yet general theoretical model to study the inherent power of an *ad hoc* network of tiny mobile devices. We introduce a new candidate model of *community protocols*. We describe what can be computed using this model, without failures and with benign failures, and highlight its capacity to tackle Byzantine failures.

2 Computation Models

After a brief and informal description of the original population protocol model in Section 2.1, we define the community protocol model in Section 2.2. In Section 2.3 we describe a version of the pointer machine model that we use as an intermediate step in emulating Turing machines by community protocols.

2.1 Population Protocols

In the population protocol model of Angluin *et al.* [2], a system consists of a collection of agents. Each agent can be in one of a finite number of possible states. Inputs to the system are encoded by assigning an input value to each agent, and this determines the agent's initial state. A computation proceeds by interactions between pairs of agents: when two agents meet, they can exchange information about their states and both agents simultaneously update their own states, according to a 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 in this model. The size of the system is not used in the algorithms, which are thus *uniform*. An execution of such an algorithm begins with a collection of agents assigned with initial values and proceeds by an infinite series of pairwise interactions. Roughly speaking, such an execution is called *fair* if every system configuration that is forever reachable is eventually reached.

For a finite alphabet A, A^* represents the set of all finite strings over A, and A^+ represents the set of all non-empty finite strings over A. Let Σ be the finite alphabet of input values for individual agents, and 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 in which a collection of $|x|$ agents are initialized with the characters comprising x eventually stabilizes to output $f(x)$. This means that, from some point of time onward, all agents output $f(x)$.

2.2 Community Protocols

Here, we extend the population protocol model to obtain our community protocol model by assigning unique identifiers to agents. Let U be an infinite set that contains a special symbol $\perp$ and all possible identifiers. An *algorithm* where agents get inputs from a finite set Σ and produce outputs from a finite set Y is specified by:

- a finite set, B, of possible basic states,
- a non-negative integer d representing the number of identifiers that can be remembered by an agent,
- an input map $\iota : \Sigma \to B$,
- an output map $\omega : B \to Y$, and
- a transition relation $\delta \subseteq Q^4$, where $Q = B \times U^d$.

The state of an agent stores a finite amount of information (an element of B), together with up to d identifiers. Thus, Q is the set of possible agent states. (For $q \in Q$ and $id \in U$, we write $id \in q$ if q stores id in one of its last d components.) 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 and the set U is infinite. However, this implies that the d slots in an agent's state intended for identifiers could store arbitrary amounts of information. To keep the agents as simple as possible, in the spirit of the population protocol model, we require that only actual agent identifiers are stored in an agent's state. To keep the model minimal, we do not allow an algorithm to make use of any other structural information about the identifiers, *i.e.*, how these identifiers are represented and stored. We thus require the following constraints on δ.

1. For any $(q_1, q_2, q_1', q_2') \in \delta$, if $id \in q_1'$ or $id \in q_2'$ then $id \in q_1$ or $id \in q_2$.
2. Let π be a permutation of U with $\pi(\perp) = \perp$. For $q = \langle b, u_1, u_2, \ldots, u_d \rangle \in Q$, let $\hat{\pi}(q) = \langle b, \pi(u_1), \pi(u_2), \ldots, \pi(u_d) \rangle$. If $(q_1, q_2, q_1', q_2') \in \delta$ and π is any permutation of U, we require that $(\hat{\pi}(q_1), \hat{\pi}(q_2), \hat{\pi}(q_1'), \hat{\pi}(q_2')) \in \delta$.

In short, the first constraint says that no transition may introduce new identifiers and the second ensures that identifiers can only be stored or compared for equality, but not manipulated in any other way. (From a practical perspective, the first requirement implies that the memory requirements of an agent can be bounded, depending on the set of identifiers actually used in the system.)

A *configuration* of the algorithm consists of a finite vector of elements from Q. Let Σ^+ be the set of all possible input strings. For a particular input string $x_1 \ldots x_n \in \Sigma^+$, an *initial configuration* of the algorithm is any 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 $U - \{\perp\}$. If $C = (q_1, \ldots, q_n)$ and $C' = (q_1', \ldots, q_n')$ are two configurations, we say that C *reaches* C' *in a single step* (denoted $C \to C'$) if there exist $i \neq j$ such that $(q_i, q_j, q'_i, q'_j) \in \delta$ and for all k different from i and j, $q'_k = q_k$.

Given an input string $x \in \Sigma^+$, an *execution* of the algorithm is an infinite sequence of configurations $C_1, C_2, \ldots$, such that C_1 is an initial configuration for x and $C_i \to C_{i+1}$ for all $i \geq 1$. In the failure-free model, an execution is called *fair* if, for each C that appears infinitely often in the execution and for each C' such that $C \to C'$, C' also appears infinitely often in the execution. (Later, we shall revisit the definition of fairness when we consider Byzantine failures.) Algorithms are only required to produce correct results in fair executions.

We say that an algorithm computes a function $f : \Sigma^+ \to Y$ if, for all input strings $x \in \Sigma^+$, every fair execution $C_1, C_2, \ldots$ starting from any initial configuration C_1 for x converges to output $f(x)$, *i.e.* there is some i such that for all $j > i$, and for every $(b, u_1, \ldots, u_d)$ appearing in C_j, $\omega(b) = f(x)$.

Any population protocol can be viewed as a community protocol that has $d = 0$.

2.3 Storage Modification Machines

We shall use a pointer machine model to characterize what can be computed by community protocols. Several related types of pointer machines were developed independently: Kolmogorov-Uspenskiĭ machines [13], linking automata [12] and storage modification machines [17]. (See [6] for a survey.) Our formalism will follow the *storage modification machine (SMM)* model defined by Schönhage, with slight revisions.

An SMM represents a single computer, not a distributed system. It 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 Δ. Any string $x \in \Delta^*$ can be used as a reference to the node that is reached from the centre of the graph by following the sequence of pointers labelled by the characters of x. This node is denoted $p^*(x)$. The basic operations of an SMM allow the machine to alter the graph by creating nodes, changing pointers and testing whether two paths of pointers lead to the same node.

An SMM is specified by a finite input alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$, a finite set of directions Δ, and a programme, which is a finite list of consecutively numbered instructions. Inputs to the SMM are finite strings from Σ^*. The following types of instructions can be used in the programme.

- **new**. This creates a new node, makes it the centre, and sets all of its outgoing pointers to the old centre.
- **recentre** x, where $x \in \Delta^+$. This changes the centre of the graph to $p^*(x)$.
- **set** $x\delta$ **to** y, where $x, y \in \Delta^*$ and $\delta \in \Delta$. This changes the pointer of node $p^*(x)$ that is labelled by δ to point to node $p^*(y)$.
- **if** $x = y$ **then goto** ℓ, where $x, y \in \Delta^*$ and ℓ is a line number. This transfers control to line ℓ if $p^*(x) = p^*(y)$.
- **input** $\ell_1, \ell_2, \ldots, \ell_r$, where $\ell_1, \ldots, \ell_r$ are line numbers. This consumes the next character of the input string and transfers control to line ℓ_i if that character is σ_i. (If there are no more input characters to be consumed, this instruction has no effect.)
- **output** o, where $o \in \{0, 1\}$. This causes the machine to halt and output o.

If an instruction on line ℓ does not specify which line to execute next, then control passes to line $\ell + 1$. Whenever a node becomes unreachable from the centre, the node can be dropped from the graph, since it will play no further role in the computation. The space complexity of an SMM is measured by the maximum number of (reachable) nodes present at any one time during the computation. Our instruction set differs slightly from Schönhage's. We have dropped input and output instructions needed for online computation, since we are concerned only with computing functions. We have given a separate name to the **recentre** instruction, because it is handled as a separate case in our proofs. We have also omitted some instructions that can be trivially implemented from ours.

It was shown by van Emde Boas [19] that an SMM can simulate any Turing machine. (A more time-efficient simulation was given later by Luginbuhl and Loui [14].)

Theorem 1 *[19] Any language decided by a Turing machine using $O(S \log S)$ space can be decided by an SMM using S nodes.*

We also introduce a *nondeterministic SMM* (NSMM). It has one additional kind of instruction:

- **choose** ℓ_0, ℓ_1, where ℓ_0 and ℓ_1 are line numbers. This instruction causes the machine to transfer control either to line ℓ_0 or to line ℓ_1 nondeterministically.

We define acceptance of a string by an NSMM in the same way as for nondeterministic Turing machines: an NSMM accepts a string x if and only if *some* computation on input x outputs 1. The following theorem is proved in exactly the same way as Theorem 1.

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

3 The Failure-Free Case

In this section, we characterize what functions can be computed by community protocols, assuming agents do not fail. We focus on Boolean-valued functions (or *predicates*). There is no real loss of generality in doing so because any function $f : \Sigma^+ \to Y$ can be computed if and only if, for every $y \in Y$, the predicate $f_y(x) = \left\{ \begin{array}{ll} 1 & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{array} \right\}$ is computable. We say a predicate f is *symmetric* if, every string y obtained by reordering the characters of a string x has $f(x) = f(y)$.

Proposition 3 *Every predicate computable by a community protocol is symmetric.*

Proof: This follows directly from the definition of computing f: Any initial configuration for input x is also an initial configuration for input x', so any legal execution that begins with input x (and stabilizes to output $f(x)$) is also a legal execution for input x'. ■

The remainder of this section is devoted to establishing the following characterization.

Theorem 4 *A predicate is computable by a community protocol if and only if it is symmetric and can be computed by a non-deterministic Turing machine using $O(n \log n)$ space.*

3.1 Simulating Community Protocols on Turing Machines

We begin with the easier direction, which shows that community protocols can be simulated by nondeterministic Turing machines. The following theorem can be proved in a way that is analogous to the proof that all predicates that are computable by population protocols are in NL [2].

Theorem 5 *If f is computable by a community protocol, then $\{x \in \Sigma^* : f(x) = 1\} \in NSPACE(n \log n)$.*

Proof: Let A be a community protocol that computes the function f. Let $x \in \Sigma^+$ be an input and let $n = |x|$. Consider the graph where each node represents a possible n-agent configuration of A in which all identifiers stored in agents are from the set $\{1, 2, \ldots, n, \perp\}$. There is an edge between C and C' if and only if $C \to C'$. This graph has $(|Q|n^d)^n$ nodes, so each node can be described using $O(n \log n)$ bits.

Let C_0 be an initial configuration for input x in which agents are assigned identifiers $1, 2, \ldots, n$. Then, $f(x) = 1$ if and only if there is a node C reachable from C_0 such that in every node reachable from C, every agent has output 1. To determine whether this is the case, a non-deterministic Turing machine can guess a node C and verify using $O(n \log n)$ space that there is a path from C_0 to C. Then, it must verify that C has the desired property. Since $NSPACE(n \log n)$ is closed under complement [11, 18], it suffices to show that checking whether C does *not* have the desired property can be done in $NSPACE(n \log n)$. This is done by guessing a configuration C' in which some agent does not have output 1, and checking that there is a path from C to C'. ■

3.2 Simulating Turing Machines with Community Protocols

To prove the converse of Theorem 5, we show that a community protocol can simulate a nondeterministic Turing machine. It would be fairly straightforward to simulate a Turing machine that uses $O(n)$ tape squares: agents could organize themselves into a line by having each agent store a pointer to a right neighbour, and then each agent would simulate one tape square, as in the simulation described by Angluin *et al.* [2]. Incidentally, this shows that our model can compute any symmetric function in $NSPACE(n)$ even if each agent can only remember two identifiers: its own and its right neighbour's.

Simulating a Turing machine that uses $O(n \log n)$ space is more difficult. Essentially, each agent must simulate $O(\log n)$ squares of the Turing machine's tape, but the agents must use a data structure consisting of identifiers to represent the contents of those squares. A similar approach (using pointers) has already been used to show that SMM's can simulate Turing machines [14, 19], as described in Theorems 1 and 2. The bulk of our proof is hence devoted to simulating an NSMM using a community protocol.

The NSMM model is a reasonably close match to our community protocol model, so designing a simulation is fairly natural. Each agent represents a node of the NSMM's graph data structure, and uses its capacity to store identifiers to represent the outgoing edges of that node. Then, one agent simulates the instructions of the NSMM programme, communicating with other agents when it needs to access the graph. However, there remain several obstacles to overcome in the design of such a simulation. We must deal with the non-deterministic definition of acceptance in the NSMM model, the possibility that some executions enter infinite loops, garbage collection to eliminate unreachable nodes, and the fact that the discovery of previously unknown agents (and their corresponding input values) may require the simulation to be restarted. In the proof sketch of the following theorem, we give a high-level description of our simulation. A more precise specification is given Appendix A.

Theorem 6 *If L is a symmetric language that can be decided by an NSMM whose graph has at most n nodes at any time in the computation, then the characteristic function of L is computable by a community protocol.*

Proof (sketch): Agents organize themselves into disjoint *strips*. Each strip is a singly-linked list of agents, where each agent in the strip has a pointer to its right neighbour and to the leftmost agent in the strip. Initially, each agent is in its own strip of length 1. Each strip carries out a simulation of the NSMM algorithm. When the rightmost agent of one strip meets the leftmost agent of another, they join the two strips together, and the new, larger strip begins a fresh simulation of the NSMM. Any computation that had been performed within the smaller strips is thrown away. (Some care is required here, since cycles can be formed if both ends of one strip meet opposite ends of a different strip at about the same time. Cycles formed in this way can be detected and broken fairly easily.) Eventually, there will be just one strip, containing all agents, and it is this strip's simulation of the NSMM will eventually lead to the stable computation of the output value.

We now describe how a strip simulates the NSMM. Each node in the graph data structure of the NSMM is represented by an agent in the strip. An agent that represents node v stores the identifiers of the agents that represent each of the $|\Delta|$ nodes to which v's pointers point. These fields of the agent's state are called p_δ, for $\delta \in \Delta$. Initially, the graph consists of a single node, which is the centre, and this node is represented by the leftmost agent in the strip. The agent representing the centre of the graph is responsible for carrying out the simulation of the NSMM programme. Its state has a *control* field that behaves like a programme counter, representing how far the simulation has progressed. When the strip begins its simulation of the NSMM programme, the centre agent's *control* field indicates that it should begin executing line 1 of that programme. After two strips have finished joining together, the leftmost agent of the strip is reinitialized to store the centre of the graph (with all pointers pointing to itself) and the *control* field of the agent is reset to indicate that it should begin executing line 1.

The strips are also used to organize the input string of the NSMM. The simulated NSMM reads input characters in the order that agents are arranged in the strip, from left to right. The agent representing the centre maintains a *next* field that stores the identifier of the leftmost agent in the strip whose input has not yet been consumed during the present simulation of the NSMM programme. (If the inputs of all agents in the strip have been consumed, this field is $\perp$.) Whenever a simulation begins, the agent representing the centre resets its *next* field to its own identifier, because it is the first agent in the strip.

We now describe how each NSMM instruction is simulated by the agent p that represents the centre. Whenever p completes one instruction, the *control* field of p is updated to represent the beginning of the appropriate line of the NSMM programme (either the next line, or the line indicated by a **goto** or **input** statement).

To simulate a **new** instruction, p must locate an agent that either does not represent a node or represents a node that has become unreachable from the centre. Agent p initiates a garbage collection to detect which nodes are unreachable. To do so, p places a "clear" token at the left end of its strip. This token traverses the strip, marking all nodes as unvisited. When p observes that the token has reached the right end of the strip, it initiates a depth-first search (DFS) of the graph. Each agent must only remember which of its children it has already searched. When all children of a node have been searched, the node itself is marked as having been searched. When the root p is marked as searched, the DFS is complete. Then, p waits until it finds an unvisited node q and uses this as the new centre. There will always be such a node q: the NSMM uses n nodes in the worst case, by the hypothesis of the theorem. Agent p sets q's $|\Delta|$ pointers to point to p, and transfers the necessary control information (its *control* and *next* fields) to q, thereby making q represent the new centre.

To simulate a **recentre** x instruction, p traverses the pointers described by the string x one by one. To do this, p uses another field of its state, called *current*, which stores the identifier of the agent q that represents the node that p has reached in its traversal. Also, p's *control* field remembers how many steps of the traversal it has performed. To advance one step of the traversal, p waits until it meets q, and then updates its own *current* field to the p_δ field of q, where δ is the next character of x. We remark that p and q may, in some cases, be the same agent. In that case p does not have to wait until it meets q; it can simulate the meeting with itself. When p has finished traversing the chain of pointers and has found the agent q that represents the new centre, it makes q the centre by transferring control information (as described in the previous paragraph).

To simulate a **set** $x\delta$ **to** y instruction, p first traverses the pointers represented by the string x to find the agent q representing the node whose pointer must be updated. It then traverses the pointers represented by y. Both traversals are done in the way described in the previous paragraph. Then, it changes q's p_δ field to the identifier of the agent that it found at the end of the second traversal.

To simulate an **if** $x = y$ **then goto** ℓ instruction, p traverses both paths of pointers as described above, and compares the identifiers of the agents reached. If they match, p changes its *control* field to indicate the beginning of line ℓ; otherwise p changes its control field to indicate the beginning of the next line of code.

We now describe how to simulate an **input** $\ell_1, \dots, \ell_r$ instruction. If p's *next* field is $\perp$, then all of the input characters have been consumed, and p simply goes on to execute the next instruction in the programme. Otherwise, p waits until it meets agent q whose identifier matches p's *next* field. (Again, p and q could be the same agent, in which case p internally simulates the following interaction as if it had met itself.) When p meets q, p changes its *control* field to represent the beginning of the line that corresponds to q's input value. That is, if q's input value was σ_i, p executes line ℓ_i next. Agent p also updates its *next* field to the identifier of the agent that is to the right of q in the strip to indicate that q's input character has been consumed.

The simulation of an **output** instruction depends on the value to be output. If the instruction outputs 1, then the simulation has successfully found an accepting computation, and we know that the input represented by the agents in the strip belongs to the language decided by the NSMM, and no further simulation is necessary. (However, if it later turns out that the strip does not contain all the agents, the strip will join with another, and the computation will begin anew.) If the **output** instruction produces 0, it is treated as a provisional output value, and the strip is reset to begin a new computation. It may be that there is another computation of the NSMM that leads to output 1, so this reset will allow the simulation to continue looking for it.

To simulate a nondeterministic choice, p makes use of the nondeterminism of the community protocol model. When p is supposed to simulate a **choose** ℓ_0, ℓ_1 instruction, it simply changes its *control* field to represent either the beginning of line ℓ_0 or the beginning of line ℓ_1, making the choice nondeterministically.

Some executions of the NSMM may never halt. We allow the centre agent to non-deterministically choose to abandon the current simulated execution at any time and restart it (just as it does when the simulation produces a 0 output). By the fairness guarantee, if there is an accepting execution of the NSMM, the simulation will eventually simulate it and set the *output* field of the centre agent to 1, which is then passed to all other agents in the system. If there is no accepting execution, the centre agent's (and all other agents') *output* field is reset to 0 when all agents are joined together into a single strip, and will remain so forever after. ■

Combining Theorems 2 and 6 yields the following result.

Corollary 7 *If L is a symmetric language in $NSPACE(n \log n)$, then the characteristic function of L is computable by a community protocol.*

Finally, combining Proposition 3, Theorem 5 and Corollary 7 completes the proof of Theorem 4.

4 Crash Failures

Now we consider a version of the community protocol model where a constant number of agents may experience unpredictable, but benign failures. The technique of Delporte-Gallet *et al.* [8] for handling crash failures in the standard population protocol model can be applied with identifiers as well. That work also discussed transient failures. To simplify matters, we restrict attention to crashes here, although transient failures could be handled in a similar way. This allows us to use simplified versions of their definitions. As in Section 3, we focus on computing Boolean-valued functions, without any real loss of generality.

A crash failure is modelled by a transition from a configuration C to a configuration C' obtained from C by removing one component. We consider executions in which at most c such transitions occur. (The fairness guarantee is unchanged.) In order to describe what is computable in this setting, we use the condition-based approach [15]: We characterize predicates that are computable when the predicate is defined on a *subset* of all possible input strings. This corresponds to assuming some precondition on the inputs.

Let Σ be a finite set of input characters for individual agents. For $x \in \Sigma^+$, let $Del(x, c)$ be the set of all non-empty strings that can be obtained from x by deleting up to c characters. Let $\mathcal{D}$ be a subset of Σ^+. We say that a predicate $f : \mathcal{D} \to \{0, 1\}$ is *c-robust* if, for all $x, y \in \mathcal{D}$, $Del(x, c) \cap Del(y, c) \neq \emptyset \Rightarrow f(x) = f(y)$. The following proposition follows directly from the definition.

Proposition 8 *If a predicate is computable in the community protocol model with c crash failures, then the predicate is c-robust.*

Proof: Consider a community protocol that computes the predicate. Let $x, y \in \mathcal{D}$ and suppose $z \in Del(x, c) \cap Del(y, c)$. There is a configuration C that can be reached from the initial configuration for input x by immediately crashing those agents whose input characters must be deleted from x to obtain z. Any failure-free legal execution E starting from C must eventually stabilize with output $f(x)$. The configuration C is also reachable from an initial configuration for input y by similar means. (The identifiers for the agents that do not die should be chosen as they are in C). So, E must eventually stabilize with output $f(y)$. Thus, $f(x) = f(y)$ and f is c-robust. ■

Thus, c-robustness is necessary for computing with c crashes. If the domain of a predicate that is computable without failures is restricted to satisfy c-robustness, that is sufficient to compute it in the presence of c crashes.

Theorem 9 *If a predicate $f : \mathcal{D} \to \{0, 1\}$ is symmetric, in $NSPACE(n \log n)$ and c-robust, then it can be computed in the community protocol model with c crash failures.*

Proof (sketch): Given an algorithm in the anonymous population protocol model, the construction of Delporte-Gallet *et al.* [8] describes how the agents can divide up into a constant number of groups and simulate, within each group, a run of the algorithm. A group's simulation of the algorithm is guaranteed to be correct if no process in the group fails. Using $2c+1$ groups guarantees that a majority of the simulated runs are correct for some input string obtained from the actual input string by deleting at most c characters. Thus, the majority of groups will stabilize on the correct output for the given input, and this majority value is used as the output for all agents.

The same construction can be used in our community protocol model. All of the bookkeeping required to divide agents into groups can be done with a finite amount of state information per agent (since it is possible to do it entirely within the population protocol model). The simulation is carried out by having each agent simulate the actions of a constant number of agents of the simulated protocol. Making this work in the community protocol model causes no difficulties, since a simulating agent can be given enough memory to remember the complete states of a constant number of agents in the simulated community protocol. ■

As an example, consider the predicate $y \geq x^2$, where x is the number of a's in the input and y is the number of b's. This predicate is computable in the community protocol model (but not in the population protocol model) without failures. If we restrict the domain of the predicate by adding the precondition that either $y - c \geq x^2$ or $y < (x - c)^2$, the predicate can be computed by a community protocol in the presence of c crashes. This is done by simulating the failure-free community protocol for the predicate $y \geq x^2$. The preconditions ensure that if the actual input has $y \geq x^2$, and at most c agents fail, the output will not change since $y - c \geq x^2$. Similarly, if we start with an legal input having $y < x^2$ and up to c agents fail, the output will not change since $y < (x - c)^2$. In either case the simulated protocol will still yield the correct answer.

5 Byzantine Failures

We now consider Byzantine failures. Suppose some agents in the system can behave arbitrarily badly. We first show that no meaningful distributed computation can take place in the original population protocol model (without agent identifiers) even with just one Byzantine agent. This result also carries over to our model, if the Byzantine agents can alter their identity. However, if agents cannot forge identifiers, some interesting computations are possible; we illustrate this by showing how to compute majority despite Byzantine agents.

5.1 Revisiting Transitions and Fairness

Studying Byzantine agents requires us first to modify the definition of the basic model to allow for arbitrary state changes. Since the order of agents within a configuration vector is unimportant we will assume for convenience that the Byzantine agents occupy the last t components of the vector. An execution with t Byzantine failures is an infinite sequence of configurations $C_1, C_2, C_3, \ldots$ such that each configuration has $n \geq t$ components, C_1 is an initial configuration of the protocol, and for all $i \geq 1$, either $C_i \to C_{i+1}$ or the first $n - t$ components of C_i are identical to the first $n - t$ components of C_{i+1}.

We must also alter the definition of a fair execution to exempt Byzantine agents from any fairness requirements. (In particular, a Byzantine agent might never interact with any agent.) Consider an execution $C_1, C_2, \ldots$ with t Byzantine agents. Let D_i be the first $n - t$ components of C_i. We say that 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 tolerating t failures if, in every fair execution with at most t Byzantine agents, each *correct* agent eventually stabilizes with the correct output value.

5.2 (Anonymous) Population Protocols do not Tolerate a Single Byzantine Failure

Let Σ be the finite set of possible agent inputs. Let X be a set of strings over Σ, each of length at least two. We say that a function f from X to an output set Y is *trivial* if $f(x)$ can be determined from a single input character for all strings x in the domain of f. More precisely, for any two strings x and y in the domain of f that both contain a common character, $f(x) = f(y)$. Clearly, any trivial function can be computed by a population protocol with any number of Byzantine failures since any correct agent can determine the output value purely from its own input. These trivial functions are the only ones that can be computed in the population protocol model when Byzantine failures can occur.[1]

Theorem 10 *Any function computable by a population protocol in the presence of a 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 input character a. We prove that $f(x) = f(y)$. Let $n = |x| + |y| - 1$ and $p_1, p_2, \ldots, p_n$ be agents. Let E be a fair, failure-free execution of the algorithm for the n agents where $p_1, \ldots, p_{|x|}$ are given the characters of x as inputs and $p_{|x|+1}, \ldots, p_n$ are given the characters of y, except for one copy of a, as inputs. Furthermore, assume p_1 has input a in E. (We remark that the input vector of E may or may not be in the domain of f.)

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 as well.

A symmetric argument (comparing E to an execution where $p_1, p_{|x|+1}, p_{|x|+2}, \ldots, p_n$ 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, $f(x) = f(y)$. ■

5.3 A Community Protocol for Byzantine-Resilient Majority

The proof of Theorem 10 applies verbatim to our community protocol model if Byzantine agents can make arbitrary changes to their own states, including their own identifiers. However, if we do not allow Byzantine agents to forge their identity, then we can compute non-trivial functions. More formally, the Byzantine failure model *without forging* requires that the value initially stored in the second component of each agent's state (*i.e.* the part of the state that stores the agent's unique identifier) does not change throughout the execution.

We now give one important example of a non-trivial function that can be computed in the Byzantine failure model without forging: *majority*. Each agent receives an input that is either -1 or 1. The majority problem is to output 1 if the sum of the inputs is positive and 0 otherwise. We describe how to compute this function, provided the absolute value of the sum is sufficiently large for all inputs in the domain. The algorithm will work in the presence of up to f Byzantine agents, where f is a known constant. This function is not trivial, so this shows that the model without forging is strictly stronger than the model with unrestricted Byzantine failures.

[1]If strings of length one can be in the domain of the function f, the notion of triviality becomes a bit more complicated: The value of a trivial function f on a single-character string a may differ from the value of f on other strings containing a, but only if no two-character strings containing a are in the domain. Such a function can be computed with one Byzantine failure by having each agent with input a start with output $f(a)$, and then switch to the output value of all longer strings that contain a upon meeting another agent.

The basic idea of our algorithm is for 1's and -1's to cancel each other when they meet, changing the states of both agents to 0. Eventually, only 1's or only -1's will be left, and the 0's can obtain their output value by remembering the last non-zero value they saw. The Byzantine agents introduce some difficulties. A single Byzantine agent could cancel the values stored in many correct agents. To avoid this, each agent p remembers the identifier of the agent p' with which it cancelled values. If p sees that $f+1$ other agents were cancelled by p', then at least one of them must be telling the truth, and p can conclude (as we argue below) that p' is a Byzantine agent. If this happens, p reverts to its initial value, remembering that p' is bad, and goes in search of a different agent to cancel. This limits the number of non-faulty agents that can be cancelled by Byzantine agents. If the majority in the input vector is sufficiently wide, this will not affect the outcome.

A detailed description of the algorithm, and the proof of the following theorem are given in Appendix B.

Theorem 11 *The community protocol described above correctly solves the majority problem, assuming no forging, with the precondition that the absolute value of the sum of the input values is greater than* $f(f+3)$.

6 Concluding Remarks

In distributed systems, memory words are sufficiently large to store process identifiers. Similarly, in modelling random access machines, word sizes are at least logarithmic in the size of the input. Thus, we can think of the difference between the population protocol and community protocol models as follows: population agents can store $O(1)$ *bits* and community agents can store $O(1)$ *words*. We have shown that, in mobile systems with unpredictable movement, storing $O(1)$ words per agent is far stronger than storing $O(1)$ bits per agent.

Our characterization builds on a connection with pointer machines, which we believe is interesting in its own right. Cook and Dymond [7] introduced a parallel version of pointer machines, where each computing element is a finite automaton connected to $O(1)$ others. Dymond [9] discussed a nondeterministic version of this model. The key difference between their models and community protocols is that theirs are synchronous. In addition, their machine begins with some elements structured as a tree, with the input string located at the leaves of the tree, whereas our model begins with no structure, to reflect *ad hoc* mobile networks.

To conclude, we highlight some interesting open research questions related to our model. Although our simulation of Turing machines by community protocols is very efficient in terms of space, it does not address efficiency in terms of time. If some probability distribution on interactions is known, could a simulation be designed so that the expected time to simulate each step of the Turing machine is small? The ability of community protocols to tolerate failures also raises many questions. What can be computed in a way that tolerates a non-constant number of failures? For the Byzantine case, can the preconditions required for majority be weakened? More generally, which functions can be computed with Byzantine failures?

References

[1] D. Angluin, J. Aspnes, M. Chan, M. J. Fischer, H. Jiang, and R. Peralta. Stably computable properties of network graphs. In *Proc. 1st IEEE International Conference on Distributed Computing in Sensor Systems*, volume 3560 of *LNCS*, pages 63–74, 2005.

[2] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, Mar. 2006.

[3] D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. In *Proc. 20th International Symposium on Distributed Computing*, volume 4167 of *LNCS*, pages 61–75, 2006.

[4] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing.* To appear.

[5] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing behavior in networks of nondeterministically interacting sensors. In *Proc. 9th International Conference on Principles of Distributed Systems*, 2005.

[6] A. M. Ben-Amran. Pointer machines and pointer algorithms: an annotated bibliography. Available from `http://www2.mta.ac.il/∼amirben`, 1998.

[7] S. A. Cook and P. W. Dymond. Parallel pointer machines. *Computational Complexity*, 3(1):19–30, Mar. 1993.

[8] 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*, volume 4026 of *LNCS*, pages 51–66, 2006.

[9] P. W. Dymond. On nondeterminism in parallel computation. *Theoretical Comput. Sci.*, 47(3):111–120, 1986.

[10] M. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Proc. 10th International Conference on Principles of Distributed Systems*, number 4305 in LNCS, pages 395–409, 2006.

[11] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[12] D. E. Knuth. *The Art of Computer Programming, volume 1, Fundamental Algorithms.* Addison-Wesley, 1968.

[13] A. N. Kolmogorov and V. A. Uspenskiĭ. On the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 13(4):3–28, 1958. English translation in *American Mathematical Society Translations*, Series 2, volume 29, pages 217–245, 1963.

[14] D. R. Luginbuhl and M. C. Loui. Hierarchies and space measures for pointer machines. *Information and Computation*, 104(2):253–270, June 1993.

[15] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003.

[16] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes-Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.

[17] A. Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, Aug. 1980.

[18] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, Nov. 1988.

[19] P. van Emde Boas. Space measures for storage modification machines. *Inf. Process. Lett.*, 30(2):103–110, Jan. 1989.

A Detailed Construction for Theorem 6

Here, we give a more detailed specification of the simulation that is described in the proof of Theorem 6. Without loss of generality, we assume the set of labels used for pointers is $\Delta = \{1, 2, 3, \ldots, |\Delta|\}$. The basic state of an agent contains the following fields.

- *input*: element of the finite input alphabet Σ. This field is initially equal to the input value given to the agent (and remains unchanged throughout the execution).
- *output*: Boolean output value, initially 0.
- *control*: this field represents which part of the NSMM programme is being carried out. The possible values of this field will be described below. If the node does not represent the centre of the NSMM graph, this field will be either $\perp$ or *reset*.
- *garbage*: holds a value from $\{1, 2, \ldots, |\Delta|, \mathit{clear}, \mathit{unvisited}, \mathit{visited}\}$, initially *unvisited*. This field is used for garbage collection.

The state of an agent also contains the following identifier fields.

- *id*: unique identifier assigned to the agent.
- p_i (for each $1 \leq i \leq |\Delta|$): holds the identifier of the agent that represents the node pointed to by the ith pointer of the agent's node in the NSMM graph, initially $\perp$.
- *right*: identifier of right neighbour in the logical organization of agents into strips, initially $\perp$.
- *leftmost*: identifier of leftmost agent in the agent's strip, initially equal to *id*.
- *current* and *current'*: these pointers are used to store identifiers of agents representing intermediate nodes along paths that are being traced by an NSMM instruction.
- *next*: identifier of leftmost agent in strip whose input has not been read.

The set of possible values for the *control* field of an agent depends on the NSMM programme. It always includes $\perp$, *reset* and *halt*. In addition, for each line ℓ of the NSMM programme, it includes a set of possible values for the *control* field, as follows:

- $C_{\ell,0}, C_{\ell,1}, C_{\ell,2}$, if line ℓ is a new instruction,
- $C_{\ell,0}, C_{\ell,1}, \ldots, C_{\ell,k}$, if line ℓ is **recentre** $\delta_1\delta_2\ldots\delta_k$,
- $C_{\ell,0}, C_{\ell,1}, \ldots, C_{\ell,k}, C'_{\ell,1}, C'_{\ell,2}, \ldots, C'_{\ell,j}$, if line ℓ is either **set** $\delta_1\delta_2\ldots\delta_k$ **to** $\delta'_1\delta'_2\ldots\delta'_j$ or **if** $\delta_1\delta_2\ldots\delta_k = \delta'_1\delta'_2\ldots\delta'_j$ **then goto** ℓ', and
- $C_{\ell,0}$ if line ℓ is an **input, output** or **choose** instruction.

In all cases, $C_{\ell,0}$ indicates the beginning of the simulation of line ℓ; the other values are used to indicate intermediate steps in the simulation of line ℓ. Initially, the *control* field of each agent is $C_{1,0}$, to indicate that the agent should begin its simulation from the beginning of the NSMM programme.

We now describe the state transitions that occur when two agents a and b meet. The following description is simplified if we include the possibility that an agent meets itself, *i.e.*, $a = b$. (Such an interaction can obviously be simulated by an internal action of an agent.) If any of the following rules should not be applied when an agent meets itself, we explicitly exclude it by the precondition to the rule.

Two strips meet

```
if $a.right = \perp$ and $b.leftmost = b.id \neq a.leftmost$ then
    $a.right \leftarrow b$
    $b.leftmost \leftarrow a.leftmost$
```

After strips meet, propagate new leftmost field through right strip and initiate a reset when done

```
if $a.right = b.id$ and $a.leftmost \neq b.leftmost$ then
    $b.leftmost \leftarrow a.leftmost$
    if $b.right = \perp$ then
        $b.control \leftarrow reset$
```

Reset a strip to restart the computation from scratch

```
if $a.right = b.id$ and $b.control = reset$ then
    $b.control \leftarrow \perp$
    if $a.leftmost \neq a.id$ then % propagate reset from right to left
        $a.control \leftarrow reset$
        $b.output \leftarrow 0$
    else % Reset is complete; initialize $a$ to begin running NSMM programme
        for $1 \leq i \leq |\Delta|$
            $a.p_i \leftarrow a.id$
        $a.output \leftarrow 0$
        $a.next \leftarrow a.id$
        $a.control \leftarrow C_{1,0}$
```

The method of joining strips described above has one technical problem: Suppose the right end of strip A joins the left end of strip B at around the same time as the right end of strip B joins the left end of strip A. Then we will have a ring instead of a strip. If there is ever a ring, eventually, all leftmost pointers within the ring will be identical (by fairness). In particular, some node will have its leftmost pointer pointing to itself while its right pointer is non-$\perp$. We use this fact to detect cycles and break them as follows. Breaking the cycle also induces a reset of the simulation.

Detect and break rings

```
if $a.leftmost = a.id = b.leftmost$ and $b.right = a.id$ then
    $b.control \leftarrow reset$
    $b.right \leftarrow \perp$
```

For each line ℓ in the NSMM code, we add transitions, depending on the type of instruction on line ℓ. Each of the possible NSMM instructions appear below, followed by a description of the corresponding transitions that should be included in the community protocol to simulate that line.

All the transitions below have been designed so that at most one of them can be fired at any time when two agents meet. For the most part, this is clear because the if statement depends on the control field of the centre of the strip, except for the garbage collection "subroutine" within the emulation of a **new** instruction, where the thread of control is encoded in the garbage fields of the agents to execute a depth-first search on the nodes in the graph, starting from the centre.

```
$\ell$: **new**
if $a.control = C_{\ell,0}$ and $a.leftmost = b.id$ then % initiate garbage collection within the strip
    $b.garbage \leftarrow clear$
    $a.control \leftarrow C_{\ell,1}$
if $a.garbage = clear$ and $a.right = b.id$ then % clear one more agent
    $a.garbage \leftarrow unvisited$
    $b.garbage \leftarrow clear$
if $a.control = C_{\ell,1}$ and $b.garbage = clear$ and $b.right = \perp$ then % finished clearing; start DFS
    $b.garbage \leftarrow unvisited$
    $a.garbage \leftarrow 1$
    $a.control \leftarrow C_{\ell,2}$
if $a.garbage = i$ and $1 \leq i < |\Delta|$ and $a.p_i = b.id$ and $b.garbage = visited$ then
    % search next child of $a$
    $a.garbage \leftarrow i + 1$
if $a.garbage = i$ and $1 \leq i \leq |\Delta|$ and $a.p_i = b.id$ and $b.garbage = unvisited$ then
    % start searching child $b$ of $a$
    $b.garbage \leftarrow 1$
if $a.garbage = |\Delta|$ and $a.p_{|\Delta|} = b.id$ and $b.garbage = visited$ then % finished searching $a$'s subtree
    $a.garbage \leftarrow visited$

if $a.control = C_{\ell,2}$ and $a.garbage = visited$ and $b.garbage = unvisited$ then
    % DFS is finished and we found an unused node $b$; make $b$ the new centre
    $a.control \leftarrow \perp$
    for $1 \leq i \leq |\Delta|$
        $b.p_i \leftarrow a.id$
    $b.next \leftarrow a.next$
    $b.control \leftarrow C_{\ell+1,0}$

$\ell$: **recentre** $\delta_1\delta_2 \ldots \delta_k$
if $a.control = C_{\ell,0}$ then
    $a.current \leftarrow a.p_{\delta_1}$
    $a.control \leftarrow C_{\ell,1}$
if $a.control = C_{\ell,i}$ and $1 \leq i \leq k - 1$ and $a.current = b.id$ then
    $a.current \leftarrow b.p_{\delta_{i+1}}$
    $a.control \leftarrow C_{\ell,i+1}$
if $a.control = C_{\ell,k}$ and $a.current = b.id$ then % transfer centre to $b$
    $b.next \leftarrow a.next$
    $a.control \leftarrow \perp$
    $b.control \leftarrow C_{\ell+1,0}$
```

```
$\ell$: **set** $\delta_1\delta_2\ldots\delta_k$ **to** $\delta'_1\delta'_2\ldots\delta'_j$
if $a.control = C_{\ell,0}$ and $k > 1$ then % start traversing first path
    $a.current \leftarrow a.p_{\delta_1}$
    $a.control \leftarrow C_{\ell,1}$
if $a.control = C_{\ell,i}$ and $1 \leq i \leq k-2$ and $a.current = b.id$ then % traverse first path
    $a.current \leftarrow b.p_{\delta_{i+1}}$
    $a.control \leftarrow C_{\ell,i+1}$
if $a.control = C_{\ell,k-1}$ then % reached node whose pointer must change
    if $k = 1$ then
        $a.current \leftarrow a.id$
    if $j > 0$ % start traversing second path
        $a.current' \leftarrow a.p_{\delta'_1}$
        $a.control \leftarrow C'_{\ell,1}$
    elsif $j = 0$ and $a.current = b.id$ then % set $b$'s pointer to point to $a$'s node
        $b.p_{\delta_k} \leftarrow a.id$
        $a.control \leftarrow C_{\ell+1,0}$
if $a.control = C'_{\ell,i}$ and $1 \leq i \leq j-1$ and $a.current' = b.id$ then % traverse second path
    $a.current' \leftarrow b.p_{\delta'_{i+1}}$
    $a.control \leftarrow C'_{\ell,i+1}$
if $a.control = C'_{\ell,j}$ and $a.current = b.id$ then % reached end of second path; change $b$'s node's pointer
    $b.p_{\delta_k} \leftarrow a.current'$
    $a.control \leftarrow C_{\ell+1,0}$
```

```
$\ell$: **if** $\delta_1\delta_2\ldots\delta_k = \delta'_1\delta'_2\ldots\delta'_j$ **then goto** $\ell'$
if $a.control = C_{\ell,0}$ and $k > 0$ then % start traversing first path
    $a.current \leftarrow a.p_{\delta_1}$
    $a.control \leftarrow C_{\ell,1}$
if $a.control = C_{\ell,i}$ and $1 \leq i \leq k-1$ and $a.current = b.id$ then % traverse first path
    $a.current \leftarrow b.p_{\delta_{i+1}}$
    $a.control \leftarrow C_{\ell,i+1}$
if $a.control = C_{\ell,k}$ then % reached end of first path
    if $j > 0$ % start traversing second path
        $a.current' \leftarrow a.p_{\delta'_1}$
        $a.control \leftarrow C'_{\ell,1}$
    elsif $j = 0$ and $a.current = a.id$ % test succeeds, so goto line $\ell'$
        $a.control \leftarrow C_{\ell',0}$
    elsif $j = 0$ and $a.current \neq a.id$ % test fails, so goto line $\ell + 1$
        $a.control \leftarrow C_{\ell+1,0}$
if $a.control = C'_{\ell,i}$ and $1 \leq i \leq j-1$ and $a.current' = b.id$ then %traverse second path
    $a.current' \leftarrow b.p_{\delta'_{i+1}}$
    $a.control \leftarrow C'_{\ell,i+1}$
if $a.control = C'_{\ell,j}$ then % reached end of second path
    if $a.current = a.current'$ then % test succeeds, so go to line $\ell'$
        $a.control \leftarrow C_{\ell',0}$
    else % test fails, so go to line $\ell + 1$
        $a.control \leftarrow C_{\ell+1,0}$
```

```
ℓ: input $\ell_1, \ell_2, \ldots, \ell_r$
if $a.control = C_{\ell,0}$ and $a.next = b.id$ and $b.input = \sigma_i$ % read next input character $\sigma_i$ from agent $b$
    $a.next \leftarrow b.right$
    $a.control \leftarrow C_{\ell_i,0}$
if $a.control = C_{\ell,0}$ and $a.next = \perp$ then % no more input characters
    $a.control = C_{\ell+1,0}$
```

```
ℓ: output 1
if $a.control = C_{\ell,0}$ then
    $a.output \leftarrow 1$
    $a.control \leftarrow halt$
```

```
ℓ: output 0
if $a.control = C_{\ell,0}$ and $b.leftmost = a.id$ and $b.right = \perp$ then % restart the computation
    $b.control \leftarrow reset$
```

```
ℓ: choose $\ell_0, \ell_1$
if $a.control = C_{\ell,0}$ then % make a nondeterministic choice
    $a.control \leftarrow$ either $C_{\ell_0,0}$ or $C_{\ell_1,0}$
```

Next, we add the *possibility* of resetting the computation at any time, in case the computation has entered an infinite loop. The following kind of transition is permitted as one possible (nondeterministic) option, in addition to any previously specified transition. The reset occurs when the agent representing the centre meets the rightmost agent in its strip and only if the centre has not completed a simulated execution that output 1.

```
Nondeterministic reset
if $a.control \notin \{halt, \perp, reset\}$ and $b.leftmost = a.leftmost$ and $b.right = \perp$ then
    $b.control \leftarrow reset$
```

Once the agent representing the centre has arrived at an output value, it should propagate that to all other agents.

```
Propagate output value
if $a.control \notin \{\perp, reset\}$ then
    $b.output \leftarrow a.output$
```

B Detailed Construction for Theorem 11

Here, we describe the algorithm used to prove Theorem 11 more precisely. The state of each agent contains the following fields.

- *input*: The input value ($+1$ or -1) given to the agent.
- *value*: An element of $\{-1, 0, 1\}$, initially equal to *input*.
- *id*: The agent's unique identifier.
- *canceller*: The identifier of another agent that cancelled this agent's value, initially $\perp$.

- *others*[1..f]: An array containing identifiers of up to f other agents that were cancelled by *canceller*. Initially, all entries are $\perp$.
- *bad*[1..f]: An array containing identifiers of up to f other agents that are known to be Byzantine. Initially, all entries are $\perp$.
- *view*[1..$2f+1$]: An array containing identifier-value pairs. Initially, all entries are $(\perp, 0)$.

When two agents with value fields 1 and -1 meet, and neither's identifier appears in the other's *bad* array, then the values are both set to 0, and each remembers the identifier of the other in its own *canceller* field. We call this type of interaction a *cancellation*. Now, suppose an agent p meets another agent p' with the same non-$\perp$ *canceller* field. If p's *others* array already contains the identifier of p' then p does not change state. Otherwise, p adds the identifier of p' to its *others* array, unless the array is already full. If the array is full, then p adds the identifier of p' to its *bad* array, and resets its *value*, *canceller* and *others* fields to their initial values. We call the latter type of interaction a *reset* of p.

The *view* array is used to determine the agent's output value. An agent remembers up to $2f+1$ other agents' values with a preference for storing non-zero values. More precisely, the *view* array of an agent p is updated as follows. If p meets an agent q whose identifier is already stored in p's *view*[i] field, it updates the associated value in *view*[i] to q's current *value* field. If an agent p whose *view* array contains an entry $(x, 0)$ meets an agent q with non-0 *value* whose identifier is not stored in p's *view*, then p stores q's identifier and *value* in that entry of its *view*. Finally, p should always update its own *view* field as if it had had an interaction with itself. (That is, if p's *view* contains an entry for p itself, the value associated with p in that entry is updated to p's current *value* field; otherwise an entry is added if p's new value is non-zero and some entry in p's *view* array contained a 0.) The output of an agent is the value that appears most often in its *view*.

The following lemma is crucial to showing that this algorithm works correctly.

Lemma 12 *In any execution, a non-faulty agent is reset only if its previous cancellation was with a Byzantine agent.*

Proof: Suppose the claim is false. Consider the first time, T in the execution when some non-faulty agent is reset after a cancellation with another non-faulty agent. Let these two agents be p and p'. When p is reset at time T, it has met $f+1$ other agents whose canceller fields contain p'. At most f of those are Byzantine, so at least one of them is correct. However, prior to T, p' cannot have been involved in any cancellations with non-faulty agents, by the definition of T. This contradiction establishes the lemma. ■

We can now prove the correctness of the following theorem from Section 5.

Theorem 11 *The community protocol described above correctly solves the majority problem, assuming no forging, with the precondition that the absolute value of the sum of the input values is greater than $f(f+3)$.*

Proof: We say that agent p *permanently cancels* agent p' if p' is never reset after a cancellation between p and p'. By the lemma, each correct agent can permanently cancel at most one other correct agent. Each Byzantine agent can permanently cancel at most $f+1$ non-faulty agents; if it cancels $f+2$ non-faulty agents, one of them will eventually meet all $f+1$ others and be reset.

Consider any fair execution whose input has n_+ 1's and n_- -1's. Suppose $n_+ - n_- > f(f+3)$. At least $n_+ - f$ of the agents with input 1 are correct. Of these, at most $f(f+1)$ will be permanently cancelled by Byzantine agents and at most n_- will be permanently cancelled by correct agents, leaving at least $f+1$ correct agents with input 1 that are not permanently cancelled. Since each agent is reset at most f times, there is some time after which those $f+1$ agents always have value 1. Thus, eventually, no correct agent ever has value -1. Thus, eventually, every correct agent's *view* array contains at least $f+1$ 1's (since the Byzantine agents can affect at most f entries of any correct agent's *view* array), so the output of each correct agent will stabilize to 1.

Similarly, if $n_- - n_+ > f(f+3)$, all correct agents will stabilize to the output 0. ■