Chapter 5
An Introduction to Population Protocols

James Aspnes, ***Yale University, USA***
Eric Ruppert, ***York University, Canada***

5.1 Introduction

Population protocols are used as a theoretical model for a collection (or population) of tiny mobile agents that interact with one another to carry out a computation. The agents are identically programmed finite state machines. Input values are initially distributed to the agents, and pairs of agents can exchange state information with other agents when they are close together. The movement pattern of the agents is unpredictable, but subject to some fairness constraints, and computations must eventually converge to the correct output value in any schedule that results from that movement. This framework can be used to model mobile ad hoc networks of tiny devices or collections of molecules undergoing chemical reactions. This chapter surveys results that describe what can be computed in various versions of the population protocol model.

First, consider the basic population protocol model as a starting point. A formal definition is given in Sect. 5.2. Later sections describe how this model has been extended or modified to other situations. Some other directions in which the model could be extended as future work are also identified.

The defining features of the basic model are:

- Finite-state agents. Each agent can store only a constant number of bits in its local state (independent of the size of the population).
- Uniformity. Agents in the same state are indistinguishable and a single algorithm is designed to work for populations of any size.
- Computation by direct interaction. Agents do not send messages or share memory; instead, an *interaction* between two agents updates both of their states according to a joint transition table. The actual mechanism of such interactions is abstracted away.
- Unpredictable interaction patterns. The choice of which agents interact is made by an adversary. Agents have little control over which other agents they interact with. (In some variants of the model, the adversary may be limited to pairing only agents that are adjacent in an *interaction graph*, typically representing distance

constraints.) A global *fairness condition* is imposed on the adversary to ensure the protocol makes progress.

- Distributed inputs and outputs. The input to a population protocol is distributed across the initial states of the entire population. The output is distributed to all agents.
- Convergence rather than termination. Agents cannot, in general, detect when they have completed their computation; instead, the agents' outputs are required to converge after some finite time to a common, correct value.

The population protocol model [25] was designed to represent sensor networks consisting of very limited mobile agents with no control over their own movement. It also bears a strong resemblance to models of interacting molecules in theoretical chemistry [321,322].

The population protocol model was inspired in part by the work of Diamadi and Fischer [235] on trust propagation in a social network. The *urn automata* of [24] can be seen as a first draft of the model that retained in vestigial form several features of classical automata: instead of interacting with one another, agents could interact only with a finite-state controller, complete with input tape. The motivation given for the current model in [25] was the study of sensor networks in which passive agents were carried along by other entities; the canonical example was a flock of birds with a sensor attached to each bird. The name of the model was chosen by analogy to *population processes* [70] in probability theory.

A population protocol often looks like an amorphous soup of lost, nearly mindless, anonymous agents blown here and there at the whim of the adversary. Although individual agents lack much intelligence or control over their own destinies, the population as a whole is nonetheless capable of performing significant computations. For example, even the simplest model is capable of solving some practical, classical distributed problems like leader election, majority voting, and organizing agents into groups. Some extensions of the model are much more powerful—under some conditions, they provide the same power as a traditional computer with the same total storage capacity. Some examples of simple population protocols are given in Sect. 5.2.1.

Much of the work so far on population protocols has concentrated on characterizing what predicates (*i.e.*, boolean-valued functions) on the input values can be computed. This question has been resolved for the basic model, and studied for several different variants of the model and under various assumptions, such as a bounded-degree interaction graph or random scheduling.

The worst-case interaction graph for computation turns out to be a complete graph, since any other interaction graph can simulate a complete interaction graph by shuffling states between the nodes [25]. In a complete interaction graph, all agents with the same state are indistinguishable, and only the counts of agents in each state affect the outcome of the protocol. The set of computable predicates in most variants of the basic model for such a graph is now known to be either exactly equal to or closely related to the set of *semilinear* predicates, those definable in *first-order Presburger arithmetic* [323,696]. These results, which originally appeared in [23,25,26,29–31,230], are summarized in Sects. 5.3, 5.4, 5.5, 5.7 and 5.9.

Sometimes the structure of incomplete interaction graphs can be exploited to simulate a Turing machine, which implies that a restricted interaction graph can make the system stronger than a complete interaction graph.

Several extensions of the basic model have been considered that are intended to reflect the limitations and capabilities of practical systems more accurately. The basic model requires coordinated two-way communication between interacting agents; this assumption is relaxed in Sect. 5.4. Work on incorporating agent failures into the model are discussed in Sects. 5.7 and 5.9. Versions of the model that give agents slightly increased memory capacity are discussed in Sect. 5.8.

More recent work has concentrated on performance. Because the weak scheduling assumptions in the basic model allow the adversary to draw out a computation indefinitely, the worst-case adversary scheduler is replaced by a random scheduling assumption, where the pair of agents that interacts at each step is drawn uniformly from the population as a whole. This gives a natural notion of *time* equal to the total number of steps to convergence and *parallel time* equal to the average number of steps initiated by any one agent (essentially the total number of steps divided by the number of agents).

As with adversarial scheduling, for random scheduling the best-understood case is that of a complete interaction graph. In this case, it is possible to simulate a register machine, where subpopulations of the agents hold tokens representing the various register values in unary. It is not hard to implement register operations like addition, subtraction, and comparison by local operations between pairs of agents; with the election of a leader, one can further construct a finite-state control. The main obstacle to implementing a complete register machine is to ensure that every agent completes any needed tasks for each instruction cycle before the next cycle starts. In [25], this was handled by having the leader wait a polynomial number of steps on average before starting the next cycle, a process which gives an easy proof of polynomially-bounded error but which also gives an impractically large slowdown. Subsequent work has reduced the slowdown to polylogarithmic by using epidemics both to propagate information quickly through the population and to provide timing [27,28]. These results are described in more detail in Sect. 5.6.

5.2 The Basic Model

In the basic population protocol model, a collection of agents are each given an input value, and agents have pairwise interactions in an order determined by a scheduler, subject to some fairness guarantee. Each agent is a kind of finite state machine and the program for the system describes how the states of two agents can be updated by an interaction. The agents are reliable: no failures occur. The agents' output values change over time and must eventually converge to the correct output value for the inputs that were initially distributed to the agents.

A protocol is formally specified by

- Q, a finite set of possible states for an agent,

- Σ, a finite input alphabet,
- ι, an input map from Σ to Q, where $\iota(\sigma)$ represents the initial state of an agent whose input is σ,
- ω, an output map from Q to the output range Y, where $\omega(q)$ represents the output value of an agent in state q, and
- $\delta \subseteq Q^4$, a transition relation that describes how pairs of agents can interact.

A computation proceeds according to such a protocol as follows. The computation takes place among *n agents*, where $n \geq 2$. Each agent initially has an input value from Σ. Each agent's initial state is determined by applying ι to its input value. This determines an initial configuration for an execution. A *configuration* of the system can be described by a vector of all the agents' states. Because agents with the same state are indistinguishable in the basic model, each configuration could also be viewed as an unordered multiset of states.

An execution of a protocol proceeds from the initial configuration by interactions between pairs of agents. Suppose two agents in states q_1 and q_2 meet and have an interaction. They can change into states q_1' and q_2' as a result of the interaction if (q_1,q_2,q_1',q_2') is in the transition relation δ. Note that interactions are in general asymmetric, with one agent (q_1) acting as the *initiator* of the interaction and the other (q_2) acting as the *responder*. Another way to describe δ is to list all possible interactions using the notation $(q_1,q_2) \to (q_1',q_2')$. (By convention, there is a null transition $(q_1,q_2) \to (q_1,q_2)$ if no others are specified with (q_1,q_2) on the left hand side.) If there is only one possible transition $(q_1,q_2) \to (q_1',q_2')$ for each pair (q_1,q_2), then the protocol is *deterministic*. If C and C' are configurations, $C \to C'$ means C' can be obtained from C by a single interaction of two agents. In other words, C contains two states q_1 and q_2 and C' is obtained from C by replacing q_1 and q_2 by q_1' and q_2', where (q_1,q_2,q_1',q_2') is in δ. An *execution* of the protocol is an infinite sequence of configurations $C_0,C_1,C_2,\ldots$, where C_0 is an initial configuration and $C_i \to C_{i+1}$ for all $i \geq 0$. Thus, an execution is a sequence of snapshots of the system after each interaction occurs. In a real distributed execution, interactions between several disjoint pairs of agents could take place simultaneously, but when writing down an execution those simultaneous interactions can be ordered arbitrarily. The notation $\stackrel{*}{\to}$ represents the transitive closure of $\to$, so $C \stackrel{*}{\to} C'$ means that there is a fragment of an execution that goes from configuration C to configuration C'.

The order in which pairs of agents interact is unpredictable: think of the schedule of interactions as being chosen by an adversary, so that protocols must work correctly under any schedule the adversary may choose. In order for meaningful computations to take place, the adversarial scheduler must satisfy some restrictions; otherwise it could, for example, divide the agents into isolated groups and schedule interactions only between agents that belong to the same group.

The *fairness* condition imposed on the scheduler is quite simple to state, but is somewhat subtle. Essentially, the scheduler cannot avoid a possible step forever. More formally, if C is a configuration that appears infinitely often in an execution, and $C \to C'$, then C' must also appear infinitely often in the execution. Another way to think of this is that anything that always has the potential to occur eventually does

occur: it is equivalent to require that any configuration that is always reachable is eventually reached.

At any point during an execution of a population protocol, each agent's state determines its output at that time. If the agent is in state q, its output value is $\omega(q)$. Thus, an agent's output may change over the course of an execution. The fairness constraint allows the scheduler to behave arbitrarily for an arbitrarily long period of time, but does require that it behave nicely eventually. It is therefore natural to phrase correctness as a property to be satisfied eventually too. For example, the scheduler could schedule only interactions between agents 1 and 2, leaving the other $n-2$ agents isolated, for millions of years, and it would be unreasonable to expect any sensible output during the period when only two agents have undergone state changes. Thus, for correctness, all agents must produce the correct output (for the input values that were initially distributed to the agents) at some time in the execution and continue to do so forever after that time.

In general, the transition relation can be non-deterministic: when two agents meet there may be several possible transitions they can make. This non-determinism sometimes comes in handy when describing protocols. However, it is not a crucial assumption: using a bit of additional machinery, agents can simulate a nondeterministic transition function by exploiting the nondeterminism of the interaction schedule. (See [23] for details.)

To summarize, a protocol computes a function f that maps multisets of elements of Σ to Y if, for every such multiset I and every fair execution that starts from the initial configuration corresponding to I, the output value of every agent eventually stabilizes to $f(I)$.

5.2.1 Examples of Population Protocols

Example 5.1. Suppose each agent is given an input bit, and all agents are supposed to output the 'or' of those bits. There is a very simple protocol to accomplish this: each agent with input 0 simply outputs 1 as soon as it discovers that another agent had input 1. Formally, $\Sigma = Y = Q = \{0,1\}$ and the input and output maps are the identity functions. The only interaction in δ is $(0,1) \to (1,1)$. If all agents have input 0, no agent will ever be in state 1. If some agent has input 1 the number of agents with state 1 cannot decrease and fairness ensures that it will eventually increase to n. In both cases, all agents stabilize to the correct output value.

Example 5.2. Suppose the agents represent dancers. Each dancer is (exclusively) a leader or a follower. Consider the problem of determining whether there are more leaders than followers. Let $Y = \{0,1\}$, with 1 indicating that there are more leaders than followers. A centralized solution would count the leaders and the followers and compare the totals. A more distributed solution is to ask everyone to start dancing with a partner (who must dance the opposite role) and then see if any dancers are left without a partner. This cancellation procedure is formalized as a population protocol

with $\Sigma = \{L,F\}$ and $Q = \{L,F,0,1\}$. The input map ι is the identity, and the output map ω maps L and 1 to 1 and maps F and 0 to 0. The transitions of δ are

$$\begin{aligned}(L,F) &\to (0,0),\\ (L,0) &\to (L,1),\\ (F,1) &\to (F,0) \text{ and}\\ (0,1) &\to (0,0).\end{aligned}$$

The first rule ensures that, eventually, either no L's or no F's will remain. At that point, if there are L's remaining, the second rule ensures that all agents will eventually produce output 1. Similarly, the third rule takes care of the case where F's remain. In the case of a tie, the last rule ensures that the output stabilizes to 0.

It may not be obvious why the protocol in Example 5.2 must converge. Consider, for example, the following transitions between configurations, where in each configuration, the agents that are about to interact are underlined.

$$\{\underline{L},L,\underline{F}\} \to \{\underline{0},\underline{L},0\} \to \{\underline{1},L,\underline{0}\} \to \{0,\underline{L},\underline{0}\} \to \{\underline{0},L,\underline{1}\} \to \{0,L,0\}$$

Repeating the last four transitions over and over yields a non-converging execution in which every pair of agents interacts infinitely often. However, this execution is not fair: the configuration $\{0,L,1\}$ appears infinitely often and $\{0,L,1\} \to \{1,L,1\}$, but $\{1,L,1\}$ never appears. This is because the first two agents only interact at "inconvenient" times, *i.e.*, when the third agent is in state 0. The definition of fairness rules this out. Thus, in some ways, the definition of fairness is stronger than saying that each pair of agents must interact infinitely often. (In fact, the two conditions are incomparable, since there can be fair executions in which two agents never meet. For example, an execution where every configuration is $\{L,L,L\}$ and all interactions take place between the first two agents is fair.)

Exercise 5.1. Show the protocol of Example 5.2 converges in every fair execution.

The definition of fairness was chosen to be quite weak (although it is still strong enough to allow useful computations). Many models of mobile systems assume that the mobility patterns of the agents follow some particular probability distribution. The goal of the population protocol model is to be more general. If there is an (unknown) underlying probability distribution on the interactions, which might even vary with time, and that distribution satisfies certain independence properties and ensures that every interaction's probability is bounded away from 0, then an execution will be fair with probability 1. Thus, any protocol will converge to the correct output with probability 1. So the model captures computations that are correct with probability 1 for a wide range of probability distributions, even though the model definition does not explicitly incorporate probabilities.

Other predicates can be computed using an approach similar to Example 5.2.

Exercise 5.2. Design a population protocol to determine whether more than 60% of the dancers are leaders.

Exercise 5.3. Design a population protocol to determine whether more than 60% of the dancers dance the same role.

Some predicates, however, require a different approach.

Example 5.3. Suppose each agent is given an input from $\Sigma = \{0,1,2,3\}$. Consider the problem of computing the sum of the inputs, modulo 4. The protocol can gather the sum (modulo 4) into a single agent. Once an agent has given its value to another agent, its value becomes null, and it obtains its output value from the eventually unique agent with a non-null value. Formally, let $Q = \{0,1,2,3,\perp_0,\perp_1,\perp_2,\perp_3\}$, where $\perp_v$ represents a null value with output v. Let $\iota(v) = v$ and $\omega(v) = \omega(\perp_v) = v$ for $v = 0,1,2,3$. The transition rules of δ are $(v_1,v_2) \to (v_1 + v_2,\perp_{v_1+v_2})$ and $(v_1,\perp_{v_2}) \to (v_1,\perp_{v_1})$, where v_1 and v_2 are 0,1,2 or 3. (The addition is modulo 4.) Rules of the first type ensure that, eventually, at most one agent will have a non-null value. Since the rules maintain, as an invariant, the sum of all non-null states (modulo 4), the unique remaining non-null value will be the sum modulo 4. The second type of rule then ensures that all agents with null states eventually converge to the correct output.

In some cases, agents may know when they have converged to the correct output, but in general they cannot. While computing the 'or' of input bits (Example 5.1), any agent in state 1 knows that its state will never change again: it has converged to its final output value. However, no agent in the protocol of Example 5.3 can ever be certain it has converged, since it may be that one agent with input 1 has not yet taken part in any interactions, and when it does start taking part the output value will have to change.

Two noteworthy properties of the population protocol model are its uniformity and anonymity. A protocol is *uniform* because its specification has no dependence on the number of agents that take part. In other words, no knowledge about the number of agents is required by the protocol. The system is *anonymous* because the agents are not equipped with unique identifiers and all agents are treated in the same way by the transition relation. Indeed, because the state set is finite and does not depend on the number of agents in the system, there is not even room in the state of an agent to store a unique identifier.

5.3 Computability

Just as traditional computability theory often restricts attention to decision problems, one can restrict attention to computing predicates, *i.e.*, functions with range $Y = \{0,1\}$, when studying what functions are computable by population protocols. There is no real loss of generality in this restriction. For any function f with range Y, let $P_{f,y}$ be a predicate defined by $P_{f,y}(x) = 1$ if and only if $f(x) = y$. Then, f is computable if and only if $P_{f,y}$ is computable for each $y \in Y$. The "only if" part of this statement is trivial. For the converse, a protocol can compute all the predicates $P_{f,y}$

in parallel, using a separate component of each agent's state for each y. (Note that it only makes sense to talk about computing a function using a population protocol if the function has a finite range Y.) This will eventually give each agent enough information to output the value of the function f.

For the basic population protocol model, there is an exact characterization of the computable predicates. To describe this characterization, some definitions and notation are required. A multiset over the input alphabet Σ can also be thought of as a vector with $d = |\Sigma|$ components, where each component is a natural number representing the multiplicity of one input character. For example, the input multiset $\{a,a,a,b,b\}$ over the input alphabet $\Sigma = \{a,b,c\}$ can be represented by the vector $(3,2,0) \in \mathbb{N}^3$. Let $(x_1,x_2,\ldots,x_d) \in \mathbb{N}^d$ be a vector that represents the input to a population protocol. Here, d is the size of the input alphabet, Σ. A *threshold predicate* is a predicate of the form $\sum_{i=1}^{d} c_i x_i < a$, where $c_1,\ldots,c_d$ and a are integer constants. A *remainder predicate* is a predicate of the form $\sum_{i=1}^{d} c_i x_i \equiv a \pmod{b}$, where $c_1,\ldots,c_d,a$ and $b > 0$ are integer constants. Angluin et al. [25] gave protocols to compute any threshold predicate or remainder predicate; the protocols are generalizations of those in Examples 5.2 and 5.3. They use the observation that addition is trivially obtained by renaming states: to compute $A+B$ from A and B, just pretend that any A or B token is really an $A+B$ token. Finally, one can compute the and or the or of two of these predicates by running the protocols for each of the basic predicates in parallel, using separate components of the agents' states, and negation simply involves relabeling the output values. Thus, population protocols can compute any predicate that is a boolean combination of remainder and threshold predicates. Surprisingly, the converse also holds: these are the *only* predicates that a population protocol can compute. This was shown for the basic model by Angluin, Aspnes, and Eisenstat [26].

Before discussing the proof of this result, there are two alternative characterizations of the computable predicates that are useful in understanding the result. These characterizations are also used in the details of the proof.

The first is that the computable predicates are precisely the *semilinear predicates*, defined as follows. A *semilinear set* is a subset of $\mathbb{N}^d$ that is a finite union of *linear sets* of the form $\{\mathbf{b} + k_1\mathbf{a}_1 + k_2\mathbf{a}_2 + \cdots + k_m\mathbf{a}_m \,|\, k_1,\ldots,k_m \in \mathbb{N}\}$, where $\mathbf{b}$ is a d-dimensional base vector, and $\mathbf{a}_1$ through $\mathbf{a}_m$ are basis vectors. See Figs. 5.1a and 5.1b for examples when $d = 2$. A *semilinear predicate* on inputs is one that is true precisely on a semilinear set. See Fig. 5.1c for an example.

To illustrate how semilinear predicates characterize computable predicates, consider the examples of the previous paragraph. Membership in the linear set S of Fig. 5.1a can be described by a boolean combination of threshold and remainder predicates: $(x_2 < 6) \wedge \neg(x_2 < 5) \wedge (x_1 \equiv 0 \pmod 2)$. Similarly, the linear set T of Fig. 5.1b can be described by $\neg(2x_1 - x_2 < 6) \wedge \neg(x_2 < 2) \wedge (2x_1 - x_2 \equiv 0 \pmod 6)$. The semilinear set $S \cup T$ of Fig. 5.1c is described by the disjunction of these two formulas.

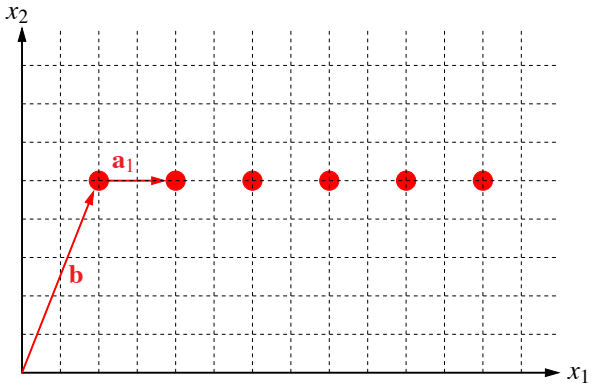

Fig. 5.1a A linear set $S = \{\mathbf{b} + k_1\mathbf{a}_1 \,|\, k_1 \in \mathbb{N}\}$

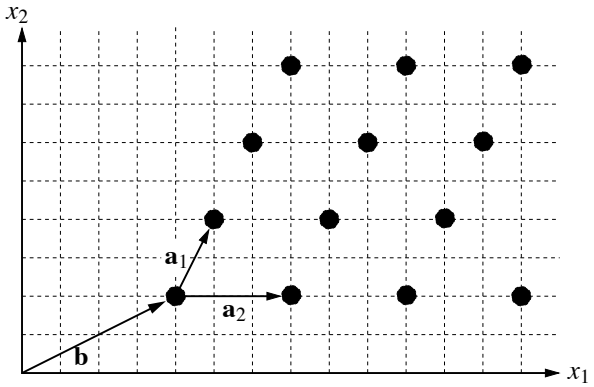

Fig. 5.1b A linear set $T = \{\mathbf{b} + k_1\mathbf{a}_1 + k_2\mathbf{a}_2 \,|\, k_1, k_2 \in \mathbb{N}\}$

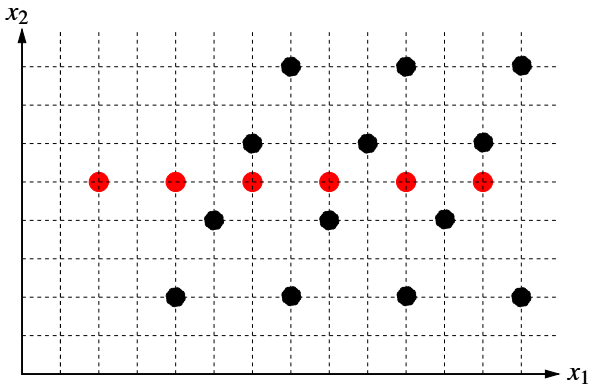

Fig. 5.1c A semilinear set $S \cup T$

A second alternative characterization of semilinear predicates is that they can be described by first-order logical formulas in Presburger arithmetic, which is arithmetic on the natural numbers with addition but not multiplication [696]. Thus, for example, the set T of Fig. 5.1b can be described by $\neg(x_1 + x_1 - x_2 < 6) \wedge \neg(x_2 < 2) \wedge \exists j(x_1 + x_1 - x_2 = j + j + j + j + j + j)$. Presburger arithmetic allows for *quantifier elimination*, replacing universal and existential quantifiers with formulas involving addition, $<$, the equivalence mod b predicates for each constant b, and the usual logical connectives $\wedge$, $\vee$, and $\neg$. For example, eliminating quantifiers from the formula for T yields $\neg(x_1 + x_1 - x_2 < 6) \wedge \neg(x_2 < 2) \wedge (x_1 + x_1 - x_2 \equiv 0 \pmod 6)$, which can be computed by a population protocol, as mentioned above.

The proof that only semilinear predicates are computable is obtained by applying results from partial order theory. The proof is quite involved, but the essential idea is that, like finite-state automata, population protocols can be "pumped" by adding extra input tokens that turn out not to affect the final output. By carefully considering exactly when this is possible, it can be shown that the positive inputs to a population protocol (considered as sets of vectors of natural numbers) can be separated into a collection of cones over some finite set of minimal positive inputs, and that each of these cones can be further expressed using only a finite set of basis vectors. This is sufficient to show that the predicate corresponds to a semilinear set as described above [26, 30]. A sketch of this argument is given in Sect. 5.3.1. The full characterization is:

Theorem 5.1 ([25,26,30]). *A predicate is computable in the basic population protocol model if and only if it is semilinear.*

Similar results with weaker classes of predicates hold for restricted models with various forms of one-way communication [29]; Sect. 5.4 describes these results in more detail. Indeed, these results were a precursor to the semilinearity theorem of [26]. The journal paper [30] combines and extends the results of [26] and [29].

A useful property of Theorem 5.1 is that it continues to hold unmodified in many simple variants of the basic model. The reason is that any change that weakens the agents can only decrease the set of computable predicates, while any model that is still strong enough to compute congruence modulo k and comparison can still compute all the semilinear predicates. So the semilinear predicates continue to be those that are computable when the inputs are not given immediately but stabilize after some finite time [23] or when one agent in an interaction can see the other's state but not vice versa [30], as in each case it is still possible to compute congruence and threshold in the limit. A similar result holds when a small number of agents can fail [230]; here a slight modification must be made to allow for partial predicates that can tolerate the loss of part of the input. All of these results are described in later sections.

5.3.1 Sketch of the Impossibility Proof

The proof that all predicates computable in the basic population protocol model are semilinear is quite technical. To give a flavor of the results, here is a simplified version, a *pumping lemma* that says that any predicate stably computed by a population protocol is a finite union of *monoids*: sets of the form

$$\{\mathbf{b} + k_1\mathbf{a}_1 + k_2\mathbf{a}_2 + \ldots \,|\, k_i \in \mathbb{N} \text{ for all } i\},$$

where the number of terms may be infinite (this is the first step in proving the full lower bound in [26], where the number of generators for each monoid is also shown to be finite). The main tool is Higman's Lemma [380], which states that any infinite sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots$ in $\mathbb{N}^d$ has elements $\mathbf{x}_i$, $\mathbf{x}_j$ with $\mathbf{x}_i \leq \mathbf{x}_j$ and $i < j$, where comparisons between vectors are done componentwise. It follows from Higman's Lemma that (a) any subset of $\mathbb{N}^d$ has finitely many minimal elements (Dickson's Lemma), and (b) any infinite subset of $\mathbb{N}^d$ contains an infinite ascending sequence $\mathbf{a}_1 < \mathbf{a}_2 < \mathbf{a}_3 \ldots$.

For the proof, configurations are represented as vectors of counts of agents in each state. Thus, a configuration is a vector in $\mathbb{N}^{|Q|}$, just as an input is a vector in $\mathbb{N}^{|\Sigma|}$. Using Dickson's Lemma, it can be shown that the set of output-stable configurations of a population protocol, where all agents agree on the output and continue to agree in all successor configurations, is semilinear. The proof is that if some configuration C is *not* output-stable, then there is some submultiset of agents $\mathbf{x}$ that can together produce an agent with a different output value. But since any $\mathbf{y} \geq \mathbf{x}$ can also produce this different output value, the property of being non-output-stable is closed upwards, implying that there is a finite collection of minimal non-output-stable configurations. Thus, the set of non-output-stable configurations is a finite union of cones, so both it and its complement—the output-stable configurations—are semilinear. (The complement of a semilinear set is also semilinear.)

Unfortunately this is not enough by itself to show that the input configurations that eventually reach a given output state are also semilinear. The second step in the argument is to show that when detecting if a configuration $\mathbf{x}$ is output-stable, it suffices to consider its truncated version

$$\tau_k(x_1, x_2, \ldots, x_m) = (\min(x_1, k), \min(x_2, k), \ldots, \min(x_m, k)),$$

provided k is large enough to encompass all of the minimal non-output-stable configurations as defined previously. The advantage of this step is that it reduces the set of configurations that must be considered from an infinite set to a finite set.

For each configuration $\mathbf{c}$ in this finite set, define the set of *extensions* $X(\mathbf{c})$ of $\mathbf{c}$ by

$$X(\mathbf{c}) = \{\mathbf{x} \,|\, \exists \mathbf{d} \text{ such that } \mathbf{c} + \mathbf{x} \xrightarrow{*} \mathbf{d} \text{ and } \tau_k(\mathbf{d}) = \tau_k(\mathbf{c})\}.$$

Intuitively, this means that $\mathbf{x}$ is in $X(\mathbf{c})$ if $\mathbf{c}$ can be "pumped" by $\mathbf{x}$, with the extra agents added in $\mathbf{x}$ disposed of in the coordinates of $\mathbf{c}$ that already have k or more

agents. It is not hard to show that extensions are composable: if $\mathbf{x},\mathbf{y}$ are in $X(\mathbf{c})$, then so is $\mathbf{x}+\mathbf{y}$. This shows that $\mathbf{b}+X(\mathbf{c})$ is a monoid for any configurations $\mathbf{b}$ and $\mathbf{c}$.

Finally, given any predicate that can be computed by a population protocol, these extensions are used to hunt for a finite collection of monoids whose union is the set Y of all inputs that produce output 1. The method is to build up a family of sets of the form $\mathbf{x}+X(\mathbf{c})$ where $\mathbf{x}$ is an input and $\mathbf{c}$ is an output-stable configuration reachable from that input. In more detail, order Y so that $\mathbf{y}_i \leq \mathbf{y}_j$ implies $i \leq j$; let $B_0 = \emptyset$; and compute B_i as follows:

- If $\mathbf{y}_i \in \mathbf{x}+X(\mathbf{c})$ for some $(\mathbf{x},\mathbf{c}) \in B_{i-1}$, let $B_i = B_{i-1}$.
- Otherwise, construct B_i by adding to B_{i-1} the pairs
  - $(\mathbf{y}_i, s(\mathbf{y}_i))$, and
  - $(\mathbf{y}_i, s(\mathbf{c}+\mathbf{y}_i-\mathbf{x}))$ for all $(\mathbf{x},\mathbf{c}) \in B_{i-1}$ with $\mathbf{x} \leq \mathbf{y}_i$,

  where $s(\mathbf{z})$ is any stable configuration reachable from $\mathbf{z}$.

Finally, let $B = \bigcup B_i$.

Then $\{\mathbf{b}+X(\mathbf{c}) \mid (\mathbf{b},\mathbf{c}) \in B\}$ covers Y because a set containing $\mathbf{y}_i$ was added to B_i if $\mathbf{y}_i$ was not already included in one of the sets in B_{i-1}. Furthermore, none of these sets contain anything outside Y. The proof of this last fact is that because $\mathbf{b} \xrightarrow{*} \mathbf{c}$, $\mathbf{z} \in \mathbf{b}+X(\mathbf{c})$ implies $\mathbf{z} \xrightarrow{*} \mathbf{z}'$ for some $\mathbf{z}' \in \mathbf{c}+X(\mathbf{c})$ (just run the $\mathbf{b} \xrightarrow{*} \mathbf{c}$ computation, ignoring any agents in $\mathbf{z}-\mathbf{b}$). But then $\mathbf{z}$ converges to same output as $\mathbf{c}$, by the definition of $X(\mathbf{c})$, the construction of B_i only includes vectors $\mathbf{c}$ that are successors of inputs in Y, so this output value is positive. It follows that B gives a representation of Y as a union of monoids, one for each element of B. It remains to show that B is finite.

To do so, suppose B is infinite. Use Higman's Lemma to get an increasing sequence $\mathbf{b}_1 < \mathbf{b}_2 < \ldots$ such that $(\mathbf{b}_i,\mathbf{c}_i) \in B$ for some $\mathbf{c}_i$. Use Higman's Lemma *again* to get an infinite subsequence $(\mathbf{b}_{i_j},\mathbf{c}_{i_j})$ where both the $\mathbf{b}$ and $\mathbf{c}$ components are increasing. Because these components are increasing, they eventually reach the bound imposed by truncation: for some i_j, $\tau_k(\mathbf{c}_{i_{(j+1)}}) = \tau_k(\mathbf{c}_{i_j})$. But then $\mathbf{b}_{i_{(j+1)}} - \mathbf{b}_{i_j}$ is in $X(\mathbf{c}_{i_j})$, so $\mathbf{b}_{i_{(j+1)}}$ cannot be in B, a contradiction.

This argument showed that any stably computable set has a finite cover by monoids of the form

$$\{\mathbf{b}+k_1\mathbf{a}_1+k_2\mathbf{a}_2+\ldots \mid k_i \in \mathbb{N} \text{ for all } i\}.$$

An immediate corollary is that any infinite stably computable set Y can be pumped: there is some $\mathbf{b}$ and $\mathbf{a}$ such that $\mathbf{b}+k\mathbf{a}$ is in Y for all $k \in \mathbb{N}$. Sadly, this is not enough to exclude some non-semilinear sets like $\{(x,y) \mid x < y\sqrt{2}\}$. However, with substantial additional work these bad cases can be excluded as well; the reader is referred to [26,30] for details.

5.4 One-way Communication

In the basic population protocol model, it is assumed that two interacting agents can simultaneously learn each other's state before updating their own states as a result of the interaction. This requires two-way communication between the two agents. Angluin et al. [30] studied several weaker interaction models where, in an interaction, information flows in one direction only. A *receiver* agent learns the state of a *sender* agent, but the sender learns nothing about the state of the receiver. The power of a system with such one-way communication depends on the precise nature of the communication mechanism.

The model is called a *transmission* model if the sender is aware that an interaction has happened (and can update its own state, although the update cannot depend on the state of the receiver). In an *observation* model, on the other hand, the sender's state is observed by the receiver, and the sender is not aware that its state has been observed. Another independent attribute is whether an interaction happens instantaneously (*immediate transmission* and *immediate observation* models) or requires some interval of time (*delayed transmission* and *delayed observation* models). The *queued transmission* model is similar to the delayed transmission model, except that receivers can temporarily refuse incoming messages so that they are not overwhelmed with more incoming information than they can handle. The queued transmission model is the closest to traditional message-passing models of distributed computing.

The weakest of these one-way models is the delayed observation model: Agents can observe other agents' input symbols to determine whether each input symbol is present in the system or not. If an agent ever sees another agent with the same input symbol as itself, it learns that there are at least two copies of that symbol, and can tell every other agent this fact. Thus, delayed observation protocols can detect whether the multiplicity of any particular input symbol is 0, 1 or at least 2, so a protocol can compute any predicate that depends only on this kind of information. Nothing else can be computed. For example there is no way for the system to determine whether some input symbol occurs with multiplicity at least 3. Intuitively, this is because there is no way to distinguish between a sequence of observations of several agents with the same input and a sequence of observations of a single agent.

The immediate observation model is slightly stronger: protocols in this model can count the number of agents with a particular input symbol, up to any constant threshold. For example, a protocol can determine whether the number of copies of input symbol a is 0,1,2,3 or more than 3. Consequently, any predicate that depends only on this kind of information can be computed. A kind of pumping lemma can be used to show that no other predicates are computable.

Angluin et al. also showed that the immediate and delayed transmission models are equivalent in power. They gave a characterization of the computable predicates that shows the power of these models is intermediate between the immediate observation model and the standard two-way model.

Finally, the queued transmission model is equivalent in power to the standard two-way model: any protocol designed for the two-way model can be simulated

using queued transmission and vice versa. This holds even though the set of configurations reachable from a particular initial configuration of a protocol in the queued transmission model is in principle unbounded; the ability to generate large numbers of buffered messages does not help the protocol, largely because there is no guarantee of where or when they will be delivered.

5.5 Restricted Interaction Graphs

In some cases the mobility of agents will have physical limitations, and this will limit the possible interactions that can occur. An *interaction graph* represents this information: nodes represent agents and edges represent possible interactions. The basic model corresponds to the case where the graph is complete. In this model, a configuration is always represented as a vector of n states. (The agents are no longer indistinguishable, so one cannot use a multiset.) If C and C' are configurations, $C \to C'$ means that C' can be obtained from C through a single interaction of *adjacent* agents, and the definitions of executions and fairness are as before, using this modified notion of a step.

Having a non-complete (but connected) interaction graph does not make the model any weaker, since adjacent agents can swap states to simulate free movement [25]. For some interaction graphs, the model becomes strictly more powerful. For example, consider a straight-line graph. It is not difficult to simulate a linear-space Turing machine by using each agent to represent one square of the Turing machine tape. This allows computation of any function or predicate that can be computed by a Turing machine using linear space. Many such functions are not semilinear and thus not computable in the complete interaction graph of the basic model. For example, a population protocol can use standard Turing machine methods to compute a multiplication predicate over the input alphabet $\{a,b,c\}$ that is true if and only if the number of a's multiplied by the number of b's is equal to the number of c's.

In addition to computing predicates on the inputs to agents, it also makes sense in this model to ask whether properties of the interaction graph itself can be computed by the agents in the system. Such problems, which were studied by Angluin et al. [23], could have useful applications in determining the network topology induced by an ad hoc deployment of mobile agents. This section describes some of their results.

As a simple example, one might want to determine whether the interaction graph has maximum degree k or more, for some fixed k. This can be done by electing a single moving leader token. Initially, all agents hold a leader token. When two leader tokens interact, the tokens coalesce, and when a leader agent interacts with a non-leader agent the leader token may change places. To test the maximum degree, the leader may instead choose to mark up to k distinct neighbors of its current node. By counting how many nodes it successfully marks, the leader can get a lower bound on the degree of the node.

A complication is that the leader has no way to detect when it has interacted with all neighbors of the current node. The best it can do is nondeterministically wait for some arbitrary but finite time before gathering in its marks and trying again. In doing so it relies on the fairness condition to eventually drive it to a state where it has correctly computed the maximum degree (or determined that it is greater than k). To accomplish the unmarking, the leader keeps track of how many marks it has placed, so that it can simply wait until it has encountered each marked neighbor again. During the initial leader election phase, two leaders deploying marks could interfere with each other. To handle this, the survivor of any interaction between two leaders collects all outstanding marks from both and resets its degree estimate.

A similar mechanism can be used to assign unique colors to all neighbors of each node in a bounded-degree graph: a wandering *colorizer* token deploys pairs of marks to its neighbors and recolors any it finds with the same color. Once this process converges, the resulting *distance-2 coloring* (so called because all nodes at distance 2 have distinct colors) effectively provides local identifiers for the neighbors of each node. These can be used to carry out arbitrary distributed computations using standard techniques (subject to the $O(1)$ space limit at each node). An example given in [23] is the construction of a rooted spanning tree, which can be used to simulate a Turing machine tape (as in the case of a line graph) by threading the Turing machine tape along a traversal of the tree (a technique described earlier for self-stabilizing systems by Itkis and Levin [417]). It follows that arbitrary properties of bounded-degree graphs that can be computed by a Turing machine using linear space can also be computed by population protocols.

5.6 Random Interactions

An alternative assumption that also greatly increases the power of the model is to replace the adversarial (but fair) scheduler of the basic model with a more constrained interaction pattern. The simplest such variant assumes *uniform random interactions*: each pair of agents is equally likely to interact at each step.

Protocols for random scheduling were given in the initial population protocol paper of Angluin et al. [25], based in part on similar protocols for the related model of urn automata [24]. The central observation was that the main limitation observed in trying to build more powerful protocols in the basic model was the inability to detect the absence of agents with a particular state. However, if a single leader agent were willing to wait long enough, it could be assured (with reasonably high probability) that it would meet every other agent in the population, and thus be able to verify the presence or absence of particular values stored in the other agents by direct inspection. The method used was to have the leader issue a single special marked token to some agent; when the leader encountered this special agent k times in a row it could be reasonably confident that the number of intervening interactions was close to $\Theta(n^{k+1})$. This is sufficient to build unary counters supporting the usual increment, decrement, and zero test operations (the last probabilistic). With counters, a regis-

ter machine with an $O(\log n)$ bit random-access memory can be simulated using a classic technique of Minsky [591].

The cost of this simulation is a polynomial blowup for the zero test and a further polynomial blowup in the simulation of the register machine. A faster simulation was given by Angluin, Aspnes, and Eisenstat [28], based on epidemics to propagate information quickly through the population. This simulation assumes a single designated leader agent in the initial configuration, which acts as the finite-state controller for the register machine. Register values are again stored in unary as tokens scattered across the remaining agents.

To execute an operation, the leader initiates an epidemic containing an operation code. This opcode is copied through the rest of the population in $\Theta(n \log n)$ interactions on average and with high probability; the latter result is shown to follow by a reduction to a concentration bound for the coupon collector problem due to Kamath et al. [451]. Arithmetic operations such as addition, comparison, subtraction, and multiplication and division by constants can be carried out by the non-leader agents in $O(n \log^c n)$ interactions (or $O(\log^c n)$ parallel time units) each, where c is a constant. Some of these algorithms are quite simple (adding A to B requires only adding a new B token to each agent that already holds an A token, possibly with an additional step of unloading extra B tokens onto empty agents to maintain $O(1)$ space per agent), while others are more involved (comparing two values in [28] involves up to $O(\log n)$ alternating rounds of doubling and cancellation, because simply having A and B tokens cancel each other as in Example 5.2 might require as many as $\Theta(n^2)$ expected interactions for the last few survivors to meet). The most expensive operation is division, at $O(n \log^5 n)$ interactions (or $O(\log^5 n)$ parallel time units).[1]

Being able to carry out individual arithmetic operations is of little use if one cannot carry out more than one. This requires that the leader be able to detect when an operation has finished, which ultimately reduces down to being able to detect when $\Theta(n \log n)$ interactions have occurred. Here the trick of issuing a single special mark is not enough, as the wait needed to ensure a low probability of premature termination is too long.

Instead, a *phase clock* based on successive waves of epidemics is used. The leader starts by initiating a phase 0 epidemic which propagates through the population in parallel to any other activity. When the leader meets an agent that is already infected with phase 0, it initiates a phase 1 epidemic that overwrites the phase 0 epidemic, and similarly with phase 2, 3, and so on, up to some fixed maximum phase $m-1$ that is in turn overwritten by phase 0 again. Angluin et al. show that, while the leader might get lucky and encounter one of a small number of newly-infected agents in a single phase, the more typical case is that a phase takes $\Theta(n \log n)$ interactions before the next is triggered, and over m phases the probability that all are too short is polynomially small. It follows that for a suitable choice of m, the phase clock gives a high-probability $\Theta(n \log n)$-interaction clock, which is enough to time the other parts of the register machine simulation.

[1] While the conference version of [28] claimed $O(n \log^4 n)$ interactions, this was the result of a calculation error that has been corrected by the authors in the full version of the paper.

A curious result in [28] is that even though the register machine simulation has a small probability of error, the same techniques can compute semilinear predicates in polylogarithmic expected parallel time with no error in the limit. The trick is to run a fast error-prone computation to get the answer quickly most of the time, and then switch to the result of a slower, error-free computation using the mechanisms of [25] after some polynomially long interval. The high time to converge for the second algorithm is apparent only when the first fails to produce the correct answer; but as this occurs only with polynomially small probability, it disappears in the expectation.

This simulation leaves room for further improvement. An immediate task is to reduce the overhead of the arithmetic operations. In [27], the same authors show how to drop the cost of the worst-case arithmetic operation to $O(n \log^2 n)$ interactions by combining a more clever register encoding with a fast *approximate majority* primitive based on dueling epidemics. This protocol has only three states: the decision values x and y, and b (for "blank"). When an x token meets a y token or vice versa, the second token turns blank. When an x or y token meets a blank agent, it converts the blank token to its own value. Much of the technical content of [27] involves showing that this process indeed converges to the majority value in $O(n \log n)$ interactions with high probability, which is done using a probabilistic potential function argument separated into several interleaved cases. The authors suggest that simplifying this argument would be a very useful target for future research. It is also possible that further improvements could reduce the overhead for arithmetic operations down to the $O(n \log n)$ interactions needed simply for all tokens to participate.

A second question is whether the distinguished leader in the initial configuration could be replaced. The coalescing leader election algorithm of [25] takes $\Theta(n^2)$ interactions to converge, which may dwarf the time for simple computations. A heuristic leader-election method is proposed in [27] that appears to converge much faster, but more analysis is needed. The authors also describe a more robust version of the phase clock of [28] that, by incorporating elements of the three-state majority protocol, appears to self-stabilize in $O(n \log n)$ interactions once the number of leaders converges to a polynomial fraction, but to date no proof of correctness for this protocol is known.

5.7 Self-stabilization and Related Problems

A series of papers [31, 32, 289] have examined the question of when population protocols can be made self-stabilizing [237], or at least can be made to tolerate input values that fluctuate over some initial part of the computation. Either condition is a stronger property than the mere convergence of the basic model, as both require that the population eventually converge to a good configuration despite an unpredictable initial configuration. Many of the algorithms designed to start in a known initial configuration (even if it is an inconvenient one, with, say, all agents in the same state) will not work if started in a particularly bad one. An example is leader election by

coalescence: this algorithm can reduce a population of many would-be leaders down to a single unique leader, but it cannot create a new leader if the initial population contains none.

Angluin et al. [31] gave the first self-stabilizing protocols for the population protocol model, showing how to carry out various tasks from previous papers without assuming a known initial configuration. These include a distance-2 coloring protocol for bounded-degree graphs based on local handshaking instead of a wandering colorizer token (which is vulnerable to being lost). Their solution has each node track whether it has interacted with a neighbor of each particular color an odd or even number of times; if a node has two neighbors of the same color, eventually its count will go out of sync with that of one or the other, causing both the node and its neighbor to choose new colors. This protocol is applied in a framework that allows self-stabilizing protocols to be composed, to give additional protocols such as rooted spanning tree construction for networks with a single special node. This last protocol is noteworthy in part because it requires $O(\log D)$ bits of storage per node, where D is the diameter of the network; it is thus one of the earliest examples of pressure to escape the restrictive $O(1)$-space assumption of the original population protocol model. Other results in this paper include a partial characterization of which network topologies do or do not support self-stabilizing leader election.

This work was continued by Angluin, Fischer, and Jiang [32], who considered the issue of solving the classic *consensus problem* [669] in an environment characterized by unpredictable communication, with the goal of converging to a common consensus value at all nodes eventually (as in a population protocol) rather than terminating with one. The paper gives protocols for solving consensus in this stabilizing sense with both crash and Byzantine failures. The model used deviates from the basic population protocol model in several strong respects: agents have identities (and the $O(\log n)$-bit memories needed to store them), and though the destinations to which messages are delivered are unpredictable, communication itself is synchronous.

Fischer and Jiang [289] return to the anonymous, asynchronous, and finite-state world of standard population protocols to consider the specific problem of leader election. As observed above, a difficulty with the simple coalescence algorithm for leader election is that it fails if there is no leader to begin with. Fischer and Jiang propose adding to the model a new *eventual leader detector*, called $\Omega?$, which acts as an oracle that eventually correctly informs the agents if there is no leader. (The name of the oracle is by analogy to the classic eventual leader *election* oracle Ω of Chandra and Toueg [164].) Self-stabilizing leader election algorithms based on $\Omega?$ are given for complete interaction graphs and rings. Curiously, the two cases distinguish between the standard global fairness condition assumed in most population protocol work and a local fairness condition that requires only that each action occurs infinitely often (but not necessarily in every configuration in which it is enabled). The latter condition is sufficient to allow self-stabilizing leader election in a complete graph but is provably insufficient in a ring. Many of these results are further elaborated in Hong Jiang's Ph.D. dissertation [433].

5.8 Larger States

The assumption that each agent can only store $O(1)$ bits of information is rather restrictive. One direction of research is to slowly relax this constraint to obtain other models that are closer to real mobile systems while still keeping the model simple enough to allow for a complete analysis.

5.8.1 Unique Identifiers

As noted in Sect. 5.2, the requirements that population protocols be independent of n and use $O(1)$ space per agent imply that agents cannot have unique identifiers. This contrasts with the vast majority of models of distributed computing, in which processes do have unique identifiers that are often a crucial component of algorithms. Guerraoui and Ruppert investigated a model, called *community protocols*, that preserve the tiny nature of agents in population protocols, but allow agents to be initially assigned unique identifiers drawn from a large set [351]. Each agent is equipped with $O(1)$ memory locations that can each store an identifier. It is assumed that transition rules cannot be dependent on the values of the identifiers: the identifiers are atomic objects that can only be tested for equality with one another. (For example, bitwise operations on identifiers are not permitted.) This preserves the property that protocols are independent of n. They gave the following precise characterization of what can be computed in this model.

Theorem 5.2 ([351]). *A predicate is computable in the community protocol model if and only if it can be computed by a nondeterministic Turing machine that uses $O(n \log n)$ space and permuting the input characters does not affect the output value.*

The necessity of the second condition (symmetry) follows immediately from the fact that the identifiers cannot be used to order the input symbols. The proof that any computable predicate can be computed using $O(n \log n)$ space on a nondeterministic Turing machine uses a nondeterministic search of the graph whose nodes are configurations of the community protocol and whose edges represent transitions between configurations.

Conversely, consider any symmetric predicate that can be computed by a nondeterministic Turing machine using $O(n \log n)$ space. The proof that it can also be computed by a community protocol uses Schönhage's pointer machines [751] as a bridge. A pointer machine is a sequential machine model that runs a program using only a directed graph structure as its memory. A community protocol can emulate a pointer machine by having each agent represent a node in the graph data structure. Some care must be taken to organize the agents to work together to simulate the sequential machine. It was known that a pointer machine that uses $O(n)$ nodes can simulate a Turing machine that uses $O(n \log n)$ space [835].

It follows that the restriction that agents can use their additional memory space only for storing $O(1)$ identifiers can essentially be overcome: the agents can do just

as much as they could if they each had $O(\log n)$ bits of storage that could be used arbitrarily.

5.8.2 Heterogeneous Systems

One interesting direction for future research is allowing some heterogeneity in the model, so that some agents have more computational power than others. As an extreme example, consider a network of weak sensors that interact with one another, but also with a base station that has unlimited capacity.

Beauquier et al. [73] studied a scenario like this, focusing on the problem of having the base station compute n, the number of mobile agents. They replaced the fairness condition of the population protocol model by a requirement that all pairs of agents interact infinitely often. They considered a self-stabilizing version of the model, where the mobile agents are initialized arbitrarily. (Otherwise the problem can be trivially solved by having the base station mark each mobile agent as it is counted.) The problem cannot be solved if each agent's memory is constant size: they proved a tight lower bound of n on the number of possible states the mobile agents must be able to store.

5.9 Failures

The work described so far assumes that the system experiences no failures. This assumption is somewhat unrealistic in the context of mobile systems of tiny agents, and was made to obtain a clean model as a starting point. Some work has studied fault-tolerant population protocols, although this topic is still largely unexplored.

5.9.1 Crash Failures

Crash failures are a relatively benign type of failure: faulty agents simply cease having any interactions at some time during the execution. Delporte-Gallet et al. [230] examined how crash failures affect the computational power of population protocols. They showed how to transform any protocol that computes a function in the failure-free model into a protocol that can tolerate $O(1)$ crash failures. However, this requires some inevitable weakening of the problem specification.

To understand how the problem specification must change when crash failures are introduced, consider the majority problem described in Example 5.2. This problem was solved under the assumption that there are no failures. Now consider a version of the majority problem where up to 5 agents may crash. Consider an execution with m followers and $m+5$ leaders. According to the original problem specification, the

output of any such execution must be 1. Suppose, however, that the agents associated with 5 of the $m+5$ leaders crash before having any interactions. There is no way that the non-faulty agents can distinguish such an execution from a failure-free execution involving m followers and m leaders. In the latter execution, the output must be 0. So, the majority problem, in its original form, cannot be solved when crash failures occur. Nevertheless, it is possible to solve a closely related problem. Suppose there are preconditions on the problem, requiring that the margin of the majority is at least 5. More precisely, it is required that either the number of leaders exceeds the number of followers by more than 5 or the number of followers exceeds the number of leaders by at least 5. Under this precondition, it can be shown that the majority problem becomes solvable even when up to 5 agents may crash.

The above example can be generalized in a natural way: to solve a problem in a way that tolerates up to f crash failures, where f is a constant, there must be a precondition that says the removal of f of the input values cannot change the output value. It is not difficult to see that such a precondition is necessary. To prove that this is sufficient to make the predicate computable in a fault-tolerant way (assuming that the original predicate is computable in the failure-free model), Delporte-Gallet et al. [230] designed an automatic transformation that converts a protocol P for the failure-free model into a protocol P' that will tolerate up to f failures.

The transformation uses replication. In P', agents are divided (in a fault-tolerant way) into $\Theta(f)$ groups, each of size $\Theta(n/f)$. Each group simulates an execution of P on the entire set of inputs. Each agent of P' can store, in its own memory, the simulated states of $O(f)$ agents of P, since f is a constant, so each group of $\Theta(n/f)$ agents has sufficient memory space to collectively simulate all agents of P. To get a group's simulation started, agents within the group gather the initial states (in P) of *all* agents. Up to f agents may crash before giving their initial states to anyone within that group, but the precondition ensures that this will not affect the output of the simulated run. Thus, any group whose members do not experience any crashes will eventually produce the correct output. It follows that at least $f+1$ of the $2f+1$ groups will converge on the correct output, and any non-faulty agent can compute this value by remembering the output value of the last agent it saw from each group and taking the majority value. (If the range of the function to be computed is larger than $\{0,1\}$, a larger number of groups must be used.)

A variant of the simulation handles a combination of a constant number of transient failures (where an agent spontaneously changes state) and crash failures [230]. It can also be used in the community protocol model described in Sect. 5.8.1 [351].

5.9.2 Byzantine Failures

An agent that has a Byzantine failure may behave arbitrarily: it can interact with all other agents, pretending to be in any state for each interaction. This behavior can cause havoc in a population protocol since none of the usual techniques used in distributed computing to identify and contain the effects of Byzantine agents can

be used. Indeed, it is known that no non-trivial predicate can be computed by a population protocol in a way that tolerates even one Byzantine agent [351]. Two ways of circumventing this fact have been studied.

In the community protocol model of Sect. 5.8.1, some failure detection is possible, provided that the agent identifiers cannot be tampered with. Guerraoui and Ruppert give a protocol that solves the majority problem, tolerating a constant number of Byzantine failures, if the margin of the majority is sufficiently wide [351]. (In defining this model, the fairness condition has to be altered to exclude Byzantine agents.)

Byzantine agents also appear in the random-scheduling work of [27], where it is shown that the approximate majority protocol quickly converges to a configuration in which nearly all non-faulty agents possess the correct decision value despite the actions of a small minority of $o(\sqrt{n})$ Byzantine agents. Here there is no extension of the basic population protocol model to include identifiers, but the convergence condition is weak, and the Byzantine agents can eventually—after exponential time—drive the protocol to any configuration, including stable configurations in which no agent holds a decision value. Determining the full power of random scheduling in the presence of Byzantine agents remains open.

5.10 Relations to Other Models

There are other mathematical models that bear some similarities to population protocols. Techniques or results from those models might prove useful in studying population protocols.

The *cellular automata* model of von Neumann [843] also models computation by a collection of communicating finite automata. However, the agents lack any mobility. In the classical version of this model, identical agents are arranged in a highly symmetric, regular, constant-degree graph (such as a grid) and each agent updates its state based on a snapshot of all of its neighbours' states. This model assumed all agents run synchronously, but some researchers have studied an asynchronous version of the model, defined by Ingerson and Buvel [414], in which a single agent updates its state in each step. The way in which this agent is chosen varies. Their work (and most that followed) is experimental. In each interaction, only one agent's state is updated (as in the immediate observation model discussed in Sect. 5.4). However, their model still assumes that an agent can learn the states of all its neighbours simultaneously, in contrast to the pairwise interactions that form the basis of population protocols.

Inspired by biological processes, Păun defined *P systems* [697] to model a collection of mobile finite-state agents that interact with one another and also with *membranes*. The membranes divide space into regions and groups of agents within a single region have interactions. The system specifies a set of interaction rules for each region. These interactions can create new agents, destroy existing agents, change agents' states, cause agents to cross a membrane into an adjacent region, or

even dissolve a membrane to merge two adjacent regions. The basic model is synchronous, and the choice of which interactions happen in each round of computation is highly constrained by priorities assigned to each rule. Unlike typical distributed models of computation, where algorithms must compute correctly in all possible executions, the emphasis here is on nondeterministic computation: there should exist a correct execution for each input. These factors make P systems a very powerful computational model. However, a simplified version of this model may be appropriate for modelling mobile systems where the algorithm has some coarse-grained control over the mobility pattern of the agents (controlling which region the agent is in, without controlling its position within that region).

For the probabilistic variants of the population protocol model, where interactions are scheduled according to a probability distribution, each configuration of the system creates a probability distribution on the set of possible successor configurations. Thus, a population protocol can be modelled as a *Markov chain* in a straightforward way. Researchers have studied some classes of Markov chains that are similar to population protocols. For example, *Markov population processes* [70, 473] (which inspired the name of population protocols), model a collection of agents finite state agents but instead of pairwise interactions, a step in the process can be either the birth or death of an agent (in some particular state), or the spontaneous change of an agent from one state to another. The probabilities of these events can depend on the relative numbers of agents in each state, but agents in the same state are treated as indistinguishable, as in population protocols. Population processes have been used to model biological populations, rumour spreading and problems in queueing theory.

5.11 Summary and Outlook

Population protocol models are a fairly recent development. Some of the most basic questions about them have been answered, but there also remain a great number of open questions. There are many ways in which the population protocol model could be further extended to open new avenues of research.

So far, work on random interactions has focussed on the uniform model, where all pairs of agents are equally likely to interact at each step. This may not be a very realistic probability distribution for many systems. One way to make it more realistic (without making it impossibly difficult to analyze) might be to look at a uniform distribution within a system with a very regular interaction graph (instead of a complete graph). For example, could the simulation of a linear-space Turing machine for bounded-degree interaction graphs (described in Sect. 5.5) be made efficient in this case? Modelling the probabilistic movement of agents explicitly would also be of great interest, but would probably require substantial technical machinery.

Existing characterizations of what can be done focus on problems that require agreement (*i.e.*, all agents stabilizing to the same output). Many problems that are

of interest in distributed computing do not have this property. Agents may need to produce different outputs (as in leader election) or the output may have to be distributed across the entire system (just as the input is distributed). For example, Angluin et al. [25] describe a simple algorithm for dividing an integer input by a constant. In this case the result cannot be represented in a single agent; instead, the number of agents that output 1 stabilizes to the answer. Other examples of problems that cannot be captured by function computation are the spanning tree construction and the node colouring algorithm described in Sect. 5.5. The problem of characterizing exactly which problems of this more general type can be solved is open. Coping with failures in those kinds of computations is also not well-understood.

The model of population protocols was intentionally designed to abstract away many genuine issues in mobile systems, to obtain a model that could be theoretically analysed. Once this model is well understood, it would be desirable to begin augmenting the model to handle some of those issues. The restriction to a constant amount of memory space per agent may be overly strict: even though the model is intended to describe extremely weak agents where one should be as parsimonious as possible with memory requirements, agents with slightly larger memory capacities could also be considered. The design of algorithms that would respect other real-world constraints on such agents is also an interesting topic: for example, how can the algorithms minimize the number of interactions required in order to preserve power to extend the lifetime of the batteries used by the agents. See also Sect. 5.8.2 for a discussion of the effects of assuming heterogeneity, a common feature of practical mobile systems.

Many known population protocols require strong assumptions about the initial configuration; for example, the register machine simulation of [28] requires an initial designated leader agent, and will generally not recover from erroneous configurations (even those reachable with low probability) in which the phase clock is corrupted. It is an interesting question of whether such protocols can be made more robust, or whether the price of high performance is vulnerability to breakdown.

Acknowledgments

James Aspnes was supported in part by NSF grant CNS-0435201. Eric Ruppert was supported in part by the Natural Sciences and Engineering Research Council of Canada. A preliminary version of this survey appeared in [36].