When Birds Die: Making Population Protocols Fault-Tolerant

Carole Delporte-Gallet[1], Hugues Fauconnier[1],
Rachid Guerraoui[2], and Eric Ruppert[3]

[1] Université Paris 7, France
[2] MIT, USA and EPFL, Switzerland
[3] York University, Canada

At vobis male sit, malae tenebrae
Orci, quae omnia bella devoratis:
tam bellum mihi passerem abstulistis. [6]

Abstract. In the population protocol model introduced by Angluin *et al.* [2], a collection of agents, which are modelled by finite state machines, move around unpredictably and have pairwise interactions. The ability of such systems to compute functions on a multiset of inputs that are initially distributed across all of the agents has been studied in the absence of failures. Here, we show that essentially the same set of functions can be computed in the presence of halting and transient failures, provided preconditions on the inputs are added so that the failures cannot immediately obscure enough of the inputs to change the outcome. We do this by giving a general-purpose transformation that makes any algorithm for the fault-free setting tolerant to failures.

1 Introduction

Consider an ad hoc mobile network in which each agent is a very simple component, such as a tiny sensor with very severe constraints on memory and power. Such systems have been envisioned, for example, in Berkeley's Smart Dust project [10]. An agent can communicate with other nearby agents through wireless communication. To make use of data collected by the agents of such a system, it is necessary to aggregate the data in some way [11, 13].

Angluin *et al.* [2] introduced the notion of a computation by a population protocol to model this situation. In their model, the computation is carried out by a collection of agents, each of which receives a piece of the input. These agents move around and information can be exchanged between two agents whenever they come into contact with each other. The goal is to ensure that every agent can eventually output the value that is to be computed (assuming a fairness condition on the sequence of interactions that occur). The agents are simple devices, and can be represented as finite state machines. The abstraction also makes absolutely minimal assumptions about the movement of the system's components. In particular, the algorithms designed for such systems cannot dictate

the movement of the agents. Can interesting computations still be performed in such a model? Angluin *et al.* showed the answer is yes, assuming no agents fail. For example, protocols exist to compute parity, majority and constant-threshold functions, as well as boolean combinations of such functions.

A motivating example for their model was a flock of birds, in which each bird carries a monitoring device that measures the bird's body temperature. The devices can signal other devices within a small distance. They showed that this sensor network could be used, for example, to determine whether at least five birds in the flock have an elevated temperature, to trigger an alert indicating that there might be an illness sweeping across the flock. In this paper, we study what happens when some of those ill birds drop dead: Can interesting computations be done in the population protocol model in a way that tolerates failures?

If malicious failures can occur, it is very difficult to do anything useful in the model: a single Byzantine agent (in collusion with the adversarial scheduler of interactions) could move around the system, driving each agent into an arbitrary reachable state by having a sequence of interactions with it. Thus, we consider two types of less catastrophic failures. A *crash failure* causes an agent to cease interacting with other agents. A *transient failure* is a momentary failure that can arbitrarily corrupt the state of an agent. The agent continues executing its algorithm correctly after the transient failure occurs. Transient failures include, as a special case, sensing failures, which cause the input to an agent to be incorrect. This is because the input is part of the state of an agent and can therefore be corrupted by the transient failure. However, transient failures are more general, since they can affect the entire state of the agent. For example, they can corrupt any partial data that the agent has collected, as well as its "programme counter" which keeps track of what part of the algorithm it is executing. (Such general transient failures might be caused by electromagnetic interference from the environment during an interaction or by soft errors due to alpha particle strikes.) We shall assume that both crash failures and transient failures can occur in an execution and that we have a known upper bound on the number of failures of each type that should be tolerated.

Clearly, some functions that can be computed without failures will be impossible to compute in a model with failures. For example, if we consider the possibility of experiencing a single halting failure, a population will not be able to compute with certainty a threshold function that is 1 if at least five of the birds are ill and 0 otherwise. Consider an execution with exactly five ailing birds, one of which dies (along with its sensor) before the bird comes into contact with any other birds. The output should be 1, but this run cannot be distinguished by any live agent from a run where there are four feverish birds and the output should be 0. However, with at most one failure, we can still distinguish whether the number of ill birds is greater than five or less than five. We discuss two ways to formalize this. We can restrict the domain of the function to be computed, by adding a precondition that the number of ill birds will either be greater than five or less than five. Alternatively, we can say that the protocol will compute the result correctly when the number of sick birds is different from five, but may

output either 0 or 1 in the case where exactly five birds are sick. We explore both approaches: the former in Sect. 5 and the latter in Sect. 6.

In short, we show that, for *any* function that can be computed by a population protocol in a failure-free environment, it is possible to design a population protocol that computes the function in a way that tolerates crash failures and transient failures, provided preconditions are added or incorrect responses are permitted for inputs that are very close to other inputs that have a different response, as described above for the example about birds.

As one might expect, we use replication to achieve fault-tolerance, but in a way that is different from traditional approaches. Given a protocol that computes a function in a failure-free environment, we run several copies of the protocol. Because of the severe limitation on the memory of each agent, we need a constant fraction of the agents to cooperate to simulate one instance of the original protocol; otherwise there would not be enough space to store the states of all of the simulated agents. We divide the agents into g groups of approximately equal size. Each group simulates one instance of the failure-free protocol by having each agent in the group simulate approximately g agents of the original protocol. The value of g is chosen to ensure that the output produced by the largest number of groups' simulations is correct.

2 Related Work

The population protocol model was introduced by Angluin, Aspnes, Diamadi, Fischer and Peralta [2]. They defined the concept of stable computation of a function in this model, focussing on stable computation of predicates, which are functions whose output is a binary value. They showed that the predicates computable in this model include all that can be expressed using Presburger arithmetic and that they are all included within the complexity class NL.

They also considered variants of the model where interactions are restricted. First, the interactions can be constrained by considering a particular communication graph, which has an edge between the nodes that represent two agents if those agents are permitted to come into contact with each other. Second, they considered a randomized version of the model, where interactions are chosen randomly and uniformly, and the output must be computed with high probability. In both cases, the power of the system is increased.

Angluin, Aspnes, Chan, Fischer, Jiang and Peralta [1] further studied the model with a non-complete communication graph. They described properties of the communication graph itself that can be computed by the agents in the system. For example, the system can determine whether it contains an odd cycle.

Angluin, Aspnes, Eisenstat and Ruppert [4] considered population protocols where the interactions between pairs of agents are one-way. Each interaction has a sender and a receiver, and the sender cannot discover any information about the receiver's state in such an interaction. Full or partial characterizations of the predicates that can be stably computed (with no failures) in several variants of this model were given.

The question of tolerating failures in the population protocol model was raised by Delporte-Gallet, Fauconnier and Guerraoui [7]. They described how an example protocol can be adapted to tolerate failures. However, their approach is not generally applicable to all population protocols.

The transient failures that we consider in this paper can corrupt the internal states of agents arbitrarily. We assume that the number of such failures is bounded. Research on self-stabilizing systems [8] assumes that *any* number of processes can have corrupted states, requiring that the system eventually return to a correct configuration. Angluin, Aspnes, Fischer and Jiang incorporated the notion of self-stabilization into the population protocol model [5]. They gave some self-stabilizing protocols for classical problems such as leader election and token passing. The types of problems they studied differ from the problems we discuss here. They concentrated on stably maintaining some property (*e.g.* having a unique leader, having a legal colouring of the communication graph), whereas we focus on computing functions of inputs initially distributed across the system. This makes it necessary for us to assume a bound on the number of transient failures, so that those inputs are not lost. Also, we are concerned with creating a general-purpose transformation that converts an arbitrary algorithm that works in the failure-free setting into a fault-tolerant algorithm.

The way we transform the specification of a problem for the failure-free population protocol model into a specification for the fault-tolerant model is, in spirit, analogous to the way such transformations have been done in traditional distributed systems. Consider for instance the seminal atomic commit problem from distributed databases [9]. In a failure-free distributed system, one would typically require a transaction to commit if and only if all servers vote "yes", *i.e.*, none detected a concurrency conflict. Such a specification is clearly impossible to implement (even in a synchronous system) if one server can fail: it is indeed impossible to distinguish an execution where all servers voted "yes" and one initially crashed, from an execution where this initially crashed server voted "no". It is thus typical to allow a transaction to sometimes abort even if all servers vote "yes" (and one of them fails or is suspected to have failed), or commit a transaction even if a minority of servers vote "no" (*e.g.*, in a replicated system).

Our approach to describing functions that can be computed in the failure-prone population protocol model is also related to the condition-based approach of Mostefaoui, Rajsbaum and Raynal [12]. They described exactly what sort of precondition must be placed on the possible inputs to the consensus problem in order for it to become solvable in an asynchronous system with f halting failures using shared read-write registers.

3 Population Protocols

Our formalization of the population protocol model is based on the work of Angluin *et al.* [2]. We present a version that assumes non-deterministic, two-way interactions can take place between any pair of agents, but also allows halting failures and transient failures. A halting failure causes an agent to cease

functioning and play no further role in the execution. A transient failure corrupts the state of an agent, but the agent otherwise follows its algorithm correctly.

Each agent in the system is modelled as a finite state machine, and algorithms must be *uniform*: each finite state machine is "programmed" identically and the programming does not depend on the number of agents in the system. This makes the model strongly *anonymous*, since there is not enough space in the state to give each agent a unique identifier.

Let X be a finite input alphabet and Y be a finite output alphabet. Each agent is provided with an input drawn from X. Since agents are essentially interchangeable, an input to the system can be thought of as a multiset of elements from X. Let $\mathcal{X}$ be a set of all multisets of elements from X. Let $\mathcal{D} \subseteq \mathcal{X}$ be the set of all input multisets that can actually occur. In general, $\mathcal{D}$ may be a proper subset of $\mathcal{X}$, since there may be preconditions on what inputs are permitted. The goal of an algorithm is to compute a function $f : \mathcal{D} \to Y$. Each agent must eventually output the value of this function for the input multiset that was initially provided to the agents.

We now describe how to specify a population protocol. Let Q be the finite set of states that each agent may take. A population protocol is defined by an input assignment $i : X \to Q$, a transition function $\delta : Q \times Q \to \mathcal{P}(Q \times Q) - \{\emptyset\}$, and an output assignment $o : Q \to Y$. (The notation $\mathcal{P}(S)$ is used to denote the power set of S.) If two agents in states q_1 and q_2 encounter each other, they can change into states q_1' and q_2', where $(q_1', q_2') \in \delta(q_1, q_2)$. Without loss of generality, assume the transition function is symmetric: $\delta(q_1, q_2) = \delta(q_2, q_1)$. The protocol is called *deterministic* if $\delta(q_1, q_2)$ is a singleton set for all $q_1, q_2 \in Q$.

Let $I \in \mathcal{D}$ be an input for the system. An *execution* of the protocol on input I is an infinite sequence of configurations, $C_0, C_1, C_2, \ldots$, each of which is a multiset of states drawn from Q. The initial configuration C_0 is the multiset $\{i(x) : x \in I\}$. The configuration C_k must be obtainable from C_{k-1} by one of the following four types of transitions:

Ordinary transition: $C_k = C_{k-1} - \{q_1, q_2\} \cup \{q_1', q_2'\}$ where $\{q_1, q_2\} \subseteq C_{k-1}$ and $(q_1', q_2') \in \delta(q_1, q_2)$.
Halting failure: $C_k = C_{k-1} - \{q\}$.
Transient failure: $C_k = C_{k-1} - \{q\} \cup \{q'\}$.
Null step: $C_k = C_{k-1}$.

The output of an agent in state q is $o(q)$. We say that the execution *stably outputs* $v \in Y$ if every agent eventually outputs v and never changes its output thereafter. Formally, this means there is an i such that for all $j > i$, $o(q) = v$ for every $q \in C_j$.

If every sequence of interactions is considered to be a possible execution in the model, it would be possible to have isolated agents that never interact with one another. So the model must incorporate a fairness guarantee. Simply requiring that every pair of agents eventually meet is insufficiently strong for some interesting protocols, since the two agents might meet only at inopportune times, when their states prevent a particular kind of interaction from happening. So the research on population protocols has assumed a stronger fairness condition. In a

fair execution, if a configuration C occurs infinitely often and a configuration C' can be reached from C by an ordinary transition, then C' occurs infinitely often. If, for example, we associate probabilities with different interactions, then an execution will be fair with probability 1. A protocol *stably computes* a function $f : \mathcal{D} \to Y$ if, for every input $I \in \mathcal{D}$, every fair execution on input I stably outputs $f(I)$.

4 The Simulation

In this section, we describe how any population protocol A that stably computes a function f in a failure-free setting can be adapted to run in a setting where a bounded number of crash and transient failures can occur. To do this, we construct an algorithm B that divides agents into groups and simulates, within each group, an execution of the original protocol A. We shall show in Sect. 5 that, if we add a precondition on the inputs, this simulation will correctly compute f. We first define the kind of precondition on the inputs that will be required.

Recall that X and Y are an input and output alphabet, $\mathcal{X}$ denotes the set of all multisets of elements from X, and $\mathcal{D} \subseteq \mathcal{X}$.

Definition 1. Let $a, b \in \mathbb{N}$. A function $f : \mathcal{X} \to Y$ is called (a, b)-*robust for* $\mathcal{D}$ if, for any input multiset $I \in \mathcal{D}$ and any input $I' \in \mathcal{X}$ that can be formed from I by removing up to a elements and then adding up to b elements, $f(I) = f(I')$.

Example 2. Let $X = Y = \{0, 1\}$. Let f be the majority function: for any multiset S of 0's and 1's, $f(S) = 1$ if and only if S contains more 1's than 0's. Let $\mathcal{D}$ be the set of all input multisets where the number of 0's differs from the number of 1's by at least k. Then f is (a, b)-robust for $\mathcal{D}$ for any parameters a and b satisfying $a + b < k$. This is because, starting from any input multiset in $\mathcal{D}$, the number of input values that would have to be added and removed to change the output of f total at least k.

Let $f : \mathcal{X} \to Y$ be any function that can be stably computed by a population protocol in the failure-free environment. We shall show that if f is $(c + t, t)$-robust for $\mathcal{D}$, then f restricted to inputs from $\mathcal{D}$ can also be stably computed in an environment where up to c crash failures and up to t transient failures may occur.

Let A be a population protocol that stably computes f in the failure-free setting. The algorithm A is specified by the state set Q_A, input and output assignment functions i_A and o_A, and the transition function δ_A. Let $Q_{init} = \{i_A(x) : x \in X\}$. We shall build an algorithm B which simulates A in a way that tolerates up to c crash failures and t transient failures. We first describe the simulation. Its correctness is argued in Sect. 5.

The fault-tolerant algorithm B will divide agents up into g groups (where g is a constant to be chosen later), and simulate the original algorithm within each group. There will be roughly n/g agents in each group, where n is the number of agents in the system. (Recall that agents do not know the value of

n.) Each of the agents that comprise a group will simulate up to $2g$ distinct agents of the original algorithm A. (For clarity, we shall hereafter refer to the agents of algorithm B as "agents", and the simulated agents of algorithm A as "threads".) No thread will be simulated by two agents in the same group (except as the result of a transient failure).

In B, each agent's state contains seven fields:

- *init* stores an initial value from Q_{init}, initialized to $i_A(x)$, where x is the input for the agent. (This field is never changed by the algorithm.)
- *joined* is a boolean variable that says whether the agent has joined a group yet. Initially, it is set to $false$.
- *group* stores a value from $\{1, 2, \ldots, g\}$, initially g, which will eventually be the name of the group this agent joins.
- *sum* will be used for a division subroutine and can take values in the range $\{0, \ldots, group - 1\}$, initially 1.
- *sim* stores a multiset of up to $2g$ elements from Q_A representing the states of the threads that the agent is simulating, initially $\emptyset$.
- $given[1..g]$ stores an array of g boolean values, with each entry initially set to $false$. This will keep track of which groups contain a thread that has been given a copy of this agent's input value.
- $output[1..g]$ stores an array of g values from Y, representing the output values from the simulations carried out by each of the g groups. It can be initialized arbitrarily.

Note that the state set of algorithm B has $|Q_{init}|g(g+1)\binom{2g+|Q_A|}{2g}2^g|Y|^g$ states, and this quantity is independent of n, the number of agents in the system, as required by the model. (The number of bits needed to represent an agent's state in the simulation is $O(g \log |Q|)$.)

The first phase of an agent's actions is devoted to assigning the agent to one of the g groups. This phase ends when the agent's *joined* field is changed to *true*. The second phase will be devoted to gathering input values from approximately g other agents and simulating, within each group, an execution of the original algorithm. We shall guarantee that each non-faulty agent's input value is eventually given to exactly one thread of exactly one agent in each group. Whenever two agents in the same group meet, they nondeterministically choose an interaction of two of their threads to simulate. In those groups that have no faulty agents, the simulation will be a faithful simulation of algorithm A, and the output of each thread within that group will eventually stabilize to the correct value. We shall choose g large enough so that agents will be able to recognize (and output) a value that is being produced by a group of agents that experienced no failures.

In phase 1, we first execute the division-by-g algorithm described by Angluin *et al.* [2] to split off, from the rest of the agents, group number g, which will contain approximately n/g agents. The remaining agents then execute a division-by-$(g-1)$ algorithm to split off group number $g-1$ (again of size roughly n/g). The remaining agents then divide by $g-2$, and so on. The *group* field of the state keeps track of which division is currently being worked on by the agent.

An agent is said to *join group* i when it sets its *joined* field to *true*, if its *group* field contains i at that time. Joining a group is an irreversible action for a non-faulty agent: once the *joined* variable is set to *true*, none of the fields *joined*, *group* or *sum* will ever change again.

To accomplish phase 1, if two agents whose $joined, group$ and *sum* fields are $(false, i, s)$ and $(false, i, s')$ with $i > 2$ meet, they transition to $(false, i-1, 1)$ and $(false, i, s+s')$ if $s + s' < i$ and to $(false, i, s+s'-i)$ and $(true, i, 0)$ if $s + s' \geq i$. We shall argue below that this has the effect of making about $1/i$ of the agents that set their group field to i eventually join group i: the *sum* field accumulates a count of agents who set their group field to i and when one count reaches i, an agent can join group i. When an agent whose group field is i has been counted (but does not join group i), it changes its group field to $i-1$. When two agents whose $joined, group$ and *sum* fields are both $(false, 2, 1)$, one transitions to $(true, 2, 0)$ and the other transitions to $(true, 1, 0)$. This has the effect of splitting the agents whose group field is set to 2; half of them join group 1 and half join group 2.

When an agent p joins group i, it sets its *sim* field to $\emptyset$ (if p's $given[i]$ field is *true*) or to $\{init\}$ (if p's $given[i]$ field is $false$). In the latter case, p also changes its $given[i]$ field to *true*. If, at any time, an agent p_1 whose value of $given[i]$ is $false$ meets another agent p_2 that has joined group i and does not have a full *sim* field, p_2 adds p_1's *init* field to its *sim* field and p_1 sets $given[i]$ to *true*. Interactions of this type will have the effect of creating, for each correct agent p (and possible some faulty ones), a thread inside the *sim* field of exactly one agent in group i initialized with the initial state that p would have in algorithm A.

Whenever two agents p_1 and p_2 that have joined the same group meet, the transition function non-deterministically chooses two elements q and q' from the union of the two agents' *sim* multisets (both elements could possibly be from the same agent's *sim* multiset) and changes the two states q and q' to a pair of states given by $\delta_A(q, q')$. If the union of the two *sim* multisets contains fewer than two elements, no state change occurs in either agent.

Whenever an agent p_1 meets an agent p_2 that has joined some group i and has a non-empty *sim* field, p_1 sets its $output[i]$ field to $o_A(q)$, where q is the first element of p_2's *sim* field. The output assignment function for B is defined by taking the element that appears with the highest multiplicity in the field *output*.

Our simulation B is non-deterministic, even if the original protocol A is deterministic: when two agents in the same group meet, they non-deterministically choose which two threads should interact. However, it is not difficult to remove this non-determinism of B by making use of the non-determinism of the order in which interactions occur, using the technique described by Angluin *et al.* [1]. Each agent stores a "choice counter" which dictates which of the finite number of possible outcomes should result from an interaction. The counters are incremented by a circulating token. However, in our model, the token could be lost when a failure occurs. So instead, we can increment the choice counter of an agent when it encounters an agent in another group.

5 Correctness

Consider an infinite fair execution $C_0, C_1, C_2, \ldots$ of the simulation B on input multiset $I \in \mathcal{D}$. We first show that eventually about n/g agents join each group. Let

$$\begin{aligned}
c_i &= \text{ the number of crash failures of agents which have } group = i \\
&\quad \text{immediately before the crash,} \\
t_i &= \text{ the number of transient failures of agents which have } group = i \\
&\quad \text{immediately before the failure,} \\
t'_i &= \text{ the number of transient failures of agents which have } group = i \\
&\quad \text{immediately after the failure,} \\
x_g(j) &= n, \\
x_i(j) &= \text{ the number of ordinary steps in } C_0, \ldots, C_j \text{ that caused an agent to} \\
&\quad \text{set its } group \text{ field to } i, \text{ for } i < g, \\
W_i(j) &= \{p \in C_j : p.group = i \text{ and } p.joined = false\}, \text{ and} \\
J_i(j) &= \{p \in C_j : p.group = i \text{ and } p.joined = true\}.
\end{aligned}$$

Note that $W_i(j)$ and $J_i(j)$ are multisets. They represent the agents in configuration C_j that are waiting to finish the division-by-i algorithm and those that have joined group i, respectively. Consider the sum $S_i(j) = i|J_i(j)| + \sum_{p \in W_i(j)} p.sum$. Initially, $S_i(0) = x_i(0)$. The only time an interaction between two agents changes this sum is when one agent sets its group field to i, which increases the value of S_i by 1. Thus, an ordinary step changes the values of S_i and x_i in the same way. If an agent's *group* value is i when it crashes, the crash decreases the value of the sum by at most i. If an agent's *group* value is i just before it experiences a transient failure, that failure can decrease the sum by at most i. If an agent's *group* value is i just after it experiences a transient failure, that failure can increase the sum by at most i, since the process's *sum* field cannot exceed its *group* field. Thus, we have

$$x_i(j) - i(c_i + t_i) \leq S_i(j) \leq x_i(j) + it'_i \tag{1}$$

If the interaction that causes the change from C_j to C_{j+1} happens because two agents in $W_i(j)$ meet, one agent is removed from $W_i(j)$ to form the set $W_i(j+1)$, and never returns to $W_i(j')$ for $j' > j$ (unless by a transient failure). So, eventually (*i.e.* for sufficiently large values of j), $W_i(j)$ will contain at most one element, so we shall have $0 \leq \sum_{p \in W_i(j)} p.sum \leq i$. So (1) implies that, eventually,

$$x_i(j) - ic_i - it_i - i \leq S_i(j) - i \leq S_i(j) - \sum_{p \in W_i(j)} p.sum = i|J_i(j)| \leq S_i(j) \leq x_i(j) + it'_i. \tag{2}$$

Dividing the bounds in (2) by i yields the following bounds on the size of $J_i(j)$.

$$x_i(j)/i - c_i - t_i - 1 \leq |J_i(j)| \leq x_i(j)/i + t'_i. \tag{3}$$

If an agent has set its *group* field to i (either by a legitimate interaction or by having a transient failure) before C_j, but did not subsequently change it to $i-1$, then either it is still in $W_i(j)$ or $J_i(j)$, or it has failed. For sufficiently large j, $W_i(j)$ contains at most one agent, so for $i > 1$ we shall have

$$x_{i-1}(j) \geq x_i(j) + t'_i - |J_i(j)| - c_i - t_i - 1. \tag{4}$$

Combining (3) and (4) yields, for large j,

$$x_{i-1}(j) \geq x_i(j) + t'_i - (x_i(j)/i + t'_i) - c_i - t_i - 1 = x_i(j)\frac{i-1}{i} - c_i - t_i - 1. \tag{5}$$

Solving recurrence (5), using the boundary condition $x_g(j) = n$, gives us (for all i)

$$x_i(j) \geq ni/g - \sum_{\ell=i+1}^{g} (c_\ell + t_\ell + 1) \geq ni/g - c - t - g. \tag{6}$$

Finally, combining (3) and (6) yields

$$|J_i(j)| \geq \frac{n}{g} - 2c - 2t - g - 1 \geq \frac{n+t}{2g} \qquad \text{(as long as } n \geq 2g(2c+2t+g+1)+t). \tag{7}$$

This means that there will eventually be at least $\frac{n+t}{2g}$ agents in each group. We call group i *correct* if $t'_i = t_i = c_i = 0$. Note that each agent's *sim* field is big enough to simulate $2g$ threads, so each correct group will be able to simulate enough threads to handle all n agents, plus t extra, bogus threads that could be generated by transient failures. (A transient failure could cause an agent that has already given an initial value to group i to give another initial value to group i.)

Let $sim_i(j)$ be the union of all the multisets that are stored in *sim* fields of states in $J_i(j)$. Consider the interactions that take place after C_j that set the $given[i]$ field of some agent to *true*. Let $future_i(j)$ be the multiset of the values in the *init* fields of those agents. These are the values that get added to the *sim* fields of agents in group i after C_j.

Lemma 3. *For each correct group i, $future_i(j)$ will be empty eventually (i.e. for sufficiently large j).*

Proof. There will eventually be at least $\frac{n+t}{2g}$ agents that join group i and each can hold $2g$ values in its *sim* field. At most $n + t$ values will be added to these fields (in total), so each agent whose $given[i]$ field is $false$ will eventually either fail or meet an agent in group i that has enough room to take that agent's initial value. Eventually every agent's $given[i]$ field will become *true* and stay that way forever, so $future_i(j)$ will eventually become empty (and remain so forever). □

Lemma 4. *Let i be a correct group in the execution of B. There is a failure-free execution $D_0, D_1, \ldots$ of the population protocol A with input set $future_i(0)$ such that, for all j, $D_j = sim_i(j) \cup future_i(j)$.*

Proof. The only steps of B's execution that alter the multiset $sim_i(j) \cup future_i(j)$ are those involving interactions between two agents that have already joined group i and have at least two elements in total in their *sim* multisets. For each such step, two elements q_1 and q_2 in the *sim* multisets are changed to q_1' and q_2', where $(q_1', q_2') \in \delta(q_1, q_2)$. Thus, the corresponding step in the constructed execution of A is legal. All other steps of the constructed execution are null steps.

We must still show that the constructed execution is fair. Consider any configuration D that occurs infinitely often in the constructed execution. There is an infinite increasing sequence $j_1, j_2, \ldots$ such that $D = D_{j_1} = D_{j_2} = \cdots$. Let D' be a configuration that can be reached from D by some ordinary transition of A that changes two agents in states q_1 and q_2 to states q_1' and q_2'. We must show that D' occurs infinitely often in the constructed execution too.

Consider the sequence of steps $C_{j_1}, C_{j_2}, \ldots$. Since there are only a finite number of possible configurations, some configuration C must occur infinitely often in this sequence. By Lemma 3 the set $future_i(j)$ must be empty for all occurrences C, because it eventually becomes empty. So, in C, the union of the *sim* fields of agents in group i is equal to D, and therefore includes q_1 and q_2. Thus, there is an ordinary transition of the simulation B that changes q_1 and q_2 in those *sim* fields of C to q_1' and q_2' to form a new configuration C'. By the fairness property of the execution of B, C' must occur infinitely often. Note that the configuration of the constructed execution that corresponds to each of these occurrences of C' is equal to D'. So D' occurs infinitely often in the constructed execution. □

The following corollary follows immediately from the preceding lemma and the fact that A stably computes f.

Corollary 5. *Eventually, for every x in the sim field of any agent that has joined a correct group i, $o_A(x) = f(future_i(0))$.*

Now we show that the set $future_i(0)$ is sufficiently close to the input multiset I for correct groups.

Lemma 6. *For any correct group i, $future_i(0) = I \cup I^+ - I^-$ where $I^+, I^- \in \mathcal{X}$ and $|I^+| \leq t$ and $|I^-| \leq c + t$.*

Proof. Consider the multiset I^+ that contains, for each transient failure during the execution that leaves an agent in a state with the $given[i]$ field equal to $false$, the *init* field of the agent immediately after it experiences the transient failure. This set contains at most t elements. Each of the values in $I \cup I^+$ can be given to a *sim* field of an agent in group i^* at most once, since doing so changes the $given[i^*]$ field of an agent from $false$ to $true$, and it remains $true$ until the agent experiences a transient failure. Thus, $future_i(0) \subseteq I \cup I^+$.

Furthermore, as argued in the proof of Lemma 3, every value in $I \cup I^+$ will eventually be transferred to the *sim* field of some agent in group i, unless the agent holding that value experiences a failure before the transfer occurs. So, at most $c + t$ of the elements of $I \cup I^+$ are not in $future_i(0)$. Let I^- be the set of those elements. Then $|I^-| \leq c + t$, and $future_i(0) = I \cup I^+ - I^-$. □

Now, by choosing g appropriately, we can guarantee that the output produced by each agent in the simulation is the output produced by the simulated thread of some correct group, and this will be the correct output value.

Theorem 7. *If $f : \mathcal{X} \to Y$ is stably computable in an environment with no failures and f is $(c+t, t)$-robust for $\mathcal{D} \subseteq \mathcal{X}$, then $f : \mathcal{D} \to Y$ is stably computable in an environment with up to c crashes and t transient failures, provided $n \geq 2((|Y| + 2)(c + 2t) + 2)^2$.*

Proof. We use the simulation B described above, taking $g = |Y|(c+2t)+1$. The assumption that $n \geq 2((|Y| + 2)(c + 2t) + 2)^2$ guarantees that $n \geq 2g(2c + 2t + g + 1) + t$ for our choice of g, so the requirement for inequality (7) is satisfied.

Consider any execution of the simulation. By Corollary 5, there is some time after which every thread in every correct group i outputs $f(future_i(0))$. Also, there is a time after which no agent experiences a failure. After these two times have both passed, every agent will eventually meet an agent in each correct group i and store $f(future_i(0))$ in its local variable $output[i]$. Let C_j be the configuration of the execution of B when all of this has happened.

Let r be any agent. We shall show that, after C_j, r stably outputs a correct value. The most common value in r's $output[1..g]$ field occurs with multiplicity at least $c + 2t + 1$. Therefore, it is $output[i^*]$ for some *correct* group i^*, since at most $c + 2t$ groups can be incorrect. Therefore, the value that r outputs will be $f(future_{i^*}(0))$ for the correct group i^*.

Let $I' = future_{i^*}(0)$. By Lemma 6, $I' = I \cup I^+ - I^-$, where $|I^-| \leq c + t$ and $|I^+| \leq t$. By the robustness property of f, we have $f(I) = f(I')$. Thus, agent r stably outputs $f(I') = f(I)$, which is correct. □

We have shown that $(c+t, t)$-robustness is sufficient to compute the function f in an environment with c crash failures and t transient failures. We now show that a weaker robustness condition is *necessary*.

Proposition 8. *Suppose that $f : \mathcal{D} \to Y$ can be stably computed by a population protocol in an environment with up to c crash failures. Then f can be extended to the domain $\mathcal{X}$ so that $f : \mathcal{X} \to Y$ is $(c, 0)$-robust for $\mathcal{D}$.*

Proof. Let y_0 be any element of Y. Let A be a population protocol that stably computes f. We extend f to all input multisets $I \in \mathcal{X}$ as follows: if A produces a stable output in some fair, failure-free execution E_I with input I, let $f(I)$ be that output value. Otherwise, define $f(I) = y_0$. Note that this is an extension of f since, for $I \in \mathcal{D}$, A stably computes f.

We now show that the extension of f is $(c, 0)$-robust. Let $I \in \mathcal{D}$ and let $I' = I - I^-$, where $I^- \in \mathcal{X}$ and $|I^-| \leq c$. We must show that $f(I') = f(I)$.

Consider an execution of A on input I in which the agents with inputs from I^- immediately fail, and then the remaining agents execute $E_{I'}$. By the hypothesis of the proposition, this execution must stably output $f(I)$. But this execution was used to define $f(I')$, so $f(I') = f(I)$. □

There is a gap between the $(c+t, t)$-robustness condition which is sufficient to compute a function in the presence of failures (Theorem 7) and the $(c, 0)$-robustness condition that is necessary (Proposition 8). Closing this gap remains an open question. However, for systems in which there are only crash failures (*i.e.* $t = 0$) the condition of $(c, 0)$-robustness is both necessary and sufficient.

6 Computing Multivalued Functions

We now generalize the model used for stably computing functions to cover the possibility that the output is not uniquely determined by the input multiset. As before, let X and Y be finite input and output alphabets, and let $\mathcal{X}$ be the set of all multisets of elements from X. Let $F : \mathcal{X} \to \mathcal{P}(Y) - \{\emptyset\}$ be a function, where $F(I)$ represents the set of legal outputs for the input multiset $I \in \mathcal{X}$. A population protocol is defined exactly as in Sect. 3. However, we have a weaker definition of stable computation for such multi-valued functions. We say that a protocol *stably computes* F if, in every fair execution on input I, there is a time after which every agent outputs only values in $F(I)$. Notice that the output of an individual agent may oscillate forever, but it eventually stabilizes in the sense that it eventually becomes a legal output and remains so forever. Furthermore, different processes may output different values. This definition of stable computation coincides with the original one in the case where $F(I)$ is a singleton set for all I.

This formulation of stable computation for multi-valued functions allows us to describe the performance of our simulation in a different way.

Theorem 9. *Let $c, t \geq 0$. Suppose $F : \mathcal{X} \to \mathcal{P}(Y) - \{\emptyset\}$ is a multivalued function that can be stably computed in an environment with no failures. Then the function $F_{c,t} : \mathcal{X} \to \mathcal{P}(Y) - \{\emptyset\}$ defined by $F_{c,t}(I) = \bigcup_{\substack{|I^-| \leq c+t \\ |I^+| \leq c}} F(I \cup I^+ - I^-)$ is stably computable in an environment with up to c crash failures and t transient failures, provided $n \geq 2((|Y| + 2)(c + 2t) + 2)^2$.*

Proof (Sketch). We can run the simulation described in Sections 4 and 5, again taking $g = |Y|(c + 2t) + 1$. The proof is very similar to the proof of Theorem 7. Consider any execution on input I. It follows from Lemma 4 that the threads in each correct group i eventually stabilize to produce outputs in $F(future_i(0))$. Consider the portion of the execution after this has occurred and all failures have occurred, and then every agent has met some agent in each correct group. Consider any moment after all of this has occurred. For any agent r, the most common value in r's *output* field at that time appears in its $output[i^*]$ field for

some correct group i^*. Let $I' = future_{i^*}(0)$. By Lemma 6, $I' = I \cup I^+ - I^-$ where $|I^+| \leq t$ and $|I^-| \leq c+t$, so $F(I') \subseteq F_{c,t}(I)$. Thus the value that is output by r is in $F_{c,t}(I)$, as required. □

Remark: It follows from this proof that, in an execution of the simulation on input I where $c' \leq c$ crash failures and $t' \leq t$ transient failures *actually occur*, the value produced as the output will be in $F_{c',t'}(I)$. In particular, if the execution happens to be failure-free, the value produced will be in $F(I)$.

Example 10. Suppose $X = \{1\}$ and $Y = \{0, 1, \ldots, 99\}$. Let $F(I) = \{|I| \mod 100\}$. Then, $F_{1,2}(I) = \{F(I)-3, F(I)-2, F(I)-1, F(I), F(I)+1, F(I)+2\}$ (where addition is done modulo 100). Since F can be stably computed in the failure-free model [2], our simulation will stably compute $F_{1,2}$ in an environment that can have up to 1 crash and 2 transient failures. Thus, it is possible to count the number of agents modulo 100, even when failures can occur, if we are satisfied with an approximate answer.

7 Concluding Remarks

If the communication graph G, which specifies which pairs of agents can come into contact with each other, is not complete, our simulation technique can be applied in a straightforward way to compute any function that can be computed in the complete graph, provided G is $(c+1)$-connected so that c crashes cannot disconnect the graph. This can be done by having the two agents in each interaction non-deterministically choose whether to swap states, just as in the failure-free model [2].

Angluin, Aspnes and Eisenstat have recently shown that the only predicates that are stably computable in the failure-free population protocol model are those defined by semilinear sets of inputs [3]. This might make it possible to use a somewhat streamlined version of our simulation to compute all stably computable binary predicates in a fault-tolerant way. This is because the known protocols for computing semilinear predicates have a relatively simple form.

There are a number of directions for future work on fault-tolerant population protocols. One is to close the gap between the $(c+t, t)$-robustness condition that is sufficient to compute a function and the $(c, 0)$-robustness condition that is necessary. Angluin *et al.* describe another type of function computation in the population protocol model, where the output does not come from a finite alphabet [2]. Instead of producing the output at each agent, the output is distributed across the system, just as the input is distributed. As an example, the division-by-g algorithm that we use in our construction starts with n agents and outputs 1 at n/g of them, and 0 at all others. As is shown by our construction, we can at least approximate the result of the division algorithm in a failure-prone environment. It would be interesting to characterize the set of functions that can be computed in this sense, in a fault-tolerant way, if some limited inaccuracy in the outcome is permitted.

This paper was concerned with the fundamental computability question: is it possible to do computations in the presence of failures? Another issue to examine is how much complexity increases as a result of incorporating failures into the model. In our model, the powerful adversary can delay convergence to a stable output for an arbitrarily long time by isolating some agents from one another. Thus, to measure complexity, one would have to consider a weaker adversary. One measure would be the expected time to converge (after the last transient failure), given some probability distribution on the interactions.

Acknowledgements

We thank James Aspnes for helpful conversations. This research was funded by the Natural Sciences and Engineering Research Council of Canada, the ACI Fragile, and the Swiss National Science Foundation through NCCR-MICS.

References

1. Dana Angluin, James Aspnes, Melody Chan, Michael J. Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *Proc. International Conference on Distributed Computing in Sensor Systems*, volume 3560 of *LNCS*, pages 63–74, 2005.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 290–299, 2004. Expanded version to appear in *Distributed Computing*.
3. Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *Proc. 25th ACM Symposium on Principles of Distributed Computing*, July 2006. To appear.
4. Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. On the power of anonymous one-way communication. In *Proc. 9th International Conference on Principles of Distributed Systems*, 2005.
5. Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing behavior in networks of nondeterministically interacting sensors. In *Proc. 9th International Conference on Principles of Distributed Systems*, 2005.
6. Gaius Valerius Catullus. Carmen 3. In *Carmina*. "But curse upon you, cursed shades of Orcus, which devour all pretty things! Such a pretty sparrow you have taken away." (Transl. Francis Warre Cornish).
7. Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. What dependability for networks of mobile sensors? In *Proc. First Workshop on Hot Topics in System Dependability*, 2005.
8. Shlomi Dolev. *Self-stabilization*. MIT Press, 2000.
9. Vassos Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Proc. Workshop on Fault-Tolerant Distributed Computing*, pages 201–208, 1990.
10. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proc. 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 271–278, 1999.

11. Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, pages 131–146, 2002.
12. Achour Mostefaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6), pages 922–954, 2003.
13. Boaz Patt-Shamir. A note on efficient aggregate queries in sensor networks. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 283–289, 2004.