Fault Tolerant Network Constructors *

Othon Michail[1], Paul G. Spirakis[1,2], Michail Theofilatos[1]

[1] Department of Computer Science, University of Liverpool, UK
[2] Computer Engineering and Informatics Department, University of Patras, Greece
Email: {Othon.Michail, P.Spirakis, Michail.Theofilatos}@liverpool.ac.uk

Abstract. In this work, we consider adversarial crash faults of nodes in the network constructors model [Michail and Spirakis, 2016]. We first show that, without further assumptions, the class of graph languages that can be (stably) constructed under crash faults is non-empty but small. When there is a finite upper bound f on the number of faults, we show that it is impossible to construct any *non-hereditary* graph language and leave as an interesting open problem the *hereditary case*. On the positive side, by relaxing our requirements we prove that: (i) permitting linear waste enables to construct on $n/(2f) - f$ nodes, any graph language that is constructible in the fault-free case, (ii) *partial constructibility* (i.e., not having to generate all graphs in the language) allows the construction of a large class of graph languages. We then extend the original model with a minimal form of *fault notifications*, and our main result here is a *fault-tolerant universal constructor* that requires linear waste in the population. Finally, we show that logarithmic local memories can be exploited for a no-waste fault-tolerant simulation of any such protocol.

Keywords: network construction; distributed protocol; self stabilization; fault tolerant protocol; dynamic graph formation; fairness; self-organization;

1 Introduction and Related Work

In this work, we address the issue of the dynamic formation of graphs under faults. We do this in a minimal setting, that is, a population of agents running *Population Protocols* that can additionally activate/deactivate

* All authors were supported by the EEE/CS initiative NeST. The last author was also supported by the Leverhulme Research Centre for Functional Materials Design. This work was partially supported by the EPSRC Grant EP/P02002X/1 on Algorithmic Aspects of Temporal Graphs.

links when they meet. This model, called *Network Constructors*, was introduced in [MS16], and is based on the *Population Protocol* (PP) model [AAD+06, AAER07] and the *Mediated Population Protocol* (MPP) model [MCS11]. We are interested in answering questions like the following: If one or more faults can affect the formation process, can we always restabilize to a correct graph, and if not, what is the class of graph languages for which there exists a fault-tolerant protocol? What are the additional minimal assumptions that we need to make in order to find fault-tolerant protocols for a bigger class of languages?

Population Protocols run on networks that consist of computational entities called *agents* (or *nodes*). One of the challenging characteristics is that the agents have no control over the schedule of interactions with each other. In a population of n agents, repeatedly a pair of agents is chosen to interact. During an interaction their states are updated based on their previous states. In general, the interactions are scheduled by a *fair scheduler*. When the execution time of a protocol needs to be examined, a typical fair scheduler is the one that selects interactions uniformly at random.

Network Constructors (and its geometric variant [Mic18]) is a theoretical model that may be viewed as a minimal model for programmable matter operating in a dynamic environment [MS17]. Programmable matter refers to any type of matter that can *algorithmically* transform its physical properties, for example shape and connectivity. The transformation is the result of executing an underlying program, which can be either a centralized algorithm or a distributed protocol stored in the material itself. There is a wide range of applications, spanning from distributed robotic systems [GKR10], to smart materials, and many theoretical models (see, e.g., [DDG+14, DDG+18, MSS19, DLFS+19] and references therein), try to capture some aspects of them.

The main difference between PPs and Network Constructors is that in the PP (and the MPP) models, the focus is on computation of functions of some input values, while Network Constructors are mostly concerned with the stable formation of graphs that belong to some graph language. Fault tolerance must deal with the graph topology, thus, previous results on self-stabilizing PPs [AAFJ08, BBB13, DLFI+17, CLV+17] and MPPs [MOKY12] do not apply here.

In [MS16], Michail and Spirakis gave protocols for several basic network construction problems, and they proved several universality results by presenting generic protocols that are capable of simulating a Turing

Machine and exploiting it in order to stably construct a large class of networks, in the absence of crash failures.

In this work, we examine the setting where *adversarial crash faults* may occur, and we address the question of which families of graph languages can be stably formed. Here, adversarial crash faults mean that an adversary knows the rules of the protocol and can select some node to be removed from the population at any time. For simplicity, we assume that the faults can only happen sequentially. This means that in every step at most one fault may occur, as opposed to the case where many faults can occur during each step. These cases are equivalent in the Network Constructors model w.l.o.g., but not in the extended version of this model (which allows fault notifications) that we consider later.

A main difference between our work and traditional self-stabilization approaches is that the nodes are supplied with constant local memory, while in principle they can form linear (in the population size) number of connections per node. Existing self-stabilization approaches that are based on restarting techniques cannot be directly applied here [DIM93, Dol00], as the nodes cannot distinguish whether they still have some activated connections with the remaining nodes, after a fault has occurred. This difficulty is the reason why it is not sufficient to just reset the state of a node in case of a fault. In addition, in contrast to previous self-stabilizing approaches [GK10, DT01] that are based on *shared memory* models, two adjacent nodes can only store 1 bit of memory in the edge joining them, which denotes the existence or not of a connection between them.

Angluin *et al.* [AAFJ08] incorporated the notion of self-stabilization into the population protocol model, giving self-stabilizing protocols for some fundamental tasks such as token passing and leader election. They focused on the goal of stably maintaining some property such as having a unique leader or a legal coloring of the communication graph.

Delporte-Gallet *et al.* [DGFGR06] studied the issue of correctly computing functions on the node inputs in the Population Protocol model [AAD$^+$06], in the presence of crash faults and transient faults that can corrupt the states of the nodes. They construct a transformation which makes any protocol that works in the failure-free setting, tolerant in the presence of such failures, as long as modifying a small number of inputs does not change the output. Guerraoui and Ruppert [GR09] introduced an interesting model, called *Community Protocol*, which extends the Population Protocol model with unique identifiers and enough memory to store a constant number of other agents' identifiers. They show that this model

can solve any decision problem in NSPACE($n \log n$) while tolerating a constant number of Byzantine failures.

Peleg [Pel09] studies logical structures, constructed over static graphs, that need to satisfy the same property on the resulting structure after node or edge failures. He distinguishes between the stronger type of fault-tolerance obtained for geometric graphs (termed *rigid fault-tolerance*) and the more flexible type required for handling general graphs (termed *competitive fault-tolerance*). It differs from our work, as we address the problem of constructing such structures over dynamic graphs.

1.1 Our contribution

The goal of any Network Constructor (NET) protocol is to stabilize to a graph that belongs to (or satisfies) some graph language L, starting from an initial configuration where all nodes are in the same state and all connections are disabled. In [MS16], only the fault-free case was considered. In this work, we formally define the model that extends NETs allowing crash failures, and we examine protocols in the presence of such faults. Whenever a node crashes, it is removed from the population, along with all its activated edges. This leaves the remaining population in a state where some actions may need to be taken by the protocol in order to eventually stabilize to a correct network.

We first study the constructive power of the original NET model in the presence of crash faults. We show that the class of graph languages that is in principle constructible is non-empty but very small: for a potentially unbounded number of faults, we show that the only stably constructible language is the *Spanning Clique*. We also prove a strong impossibility result, which holds even if the size of graphs that the protocol outputs in populations of size n need only grow with n (the remaining nodes being *waste*). For a bounded number of faults, we show that any non-hereditary graph language is impossible to be constructed. However, we show that by relaxing our requirements we can extend the class of constructible graph languages. In particular, permitting linear waste enables to construct on $n/(2f) - f$ nodes, where f is a finite upper bound on the number of faults, any graph language that is constructible in a failure-free setting. Alternatively, by allowing our protocols to generate only a subset of all graphs in the language (called *partial constructibility*), a large class of graph languages becomes constructible (see Section 3).

In light of the impossibilities in the Network Constructors model, we introduce the minimal additional assumption of *fault notifications*. In

particular, after a fault on some node u occurs, all nodes that maintain an active edge with u at that time (if any) are notified. If there are no such nodes, an arbitrary node in the population is notified. In that way, we guarantee that at least one node in the population will sense the removal of u. Nevertheless, some of our constructions work without notifications in the case of a crash fault on an isolated node (Section 4).

We obtain two fault-tolerant universal constructors. One of the main technical tools that we use in them, is a fault-tolerant construction of a stable path topology (i.e., a line). We show that this topology is capable of simulating a Turing Machine (abbreviated "TM" throughout this paper), and, in the event of a fault, is capable of always reinitializing its state correctly (Section 4.1). Our protocols use a subset of the population (called *waste*) in order to construct there a TM, while the graph which belongs to the required language L is constructed in the rest of the population (called *useful space*). Throughout this paper, we call waste all nodes that do not belong to the constructed graph $G \in L$ after stabilization, and remain either isolated nodes or part of a component such as the TM. The idea is based on [MS16], where they show several universality results by constructing on k nodes of the population a network G_1 capable of simulating a TM, and then repeatedly drawing a random network G_2 on the remaining $n - k$ nodes. The idea is to execute on G_1 the TM which decides the language L with input the network G_2. If the TM accepts, it outputs G_2, otherwise the TM constructs a new random graph.

This allows a fault-tolerant construction of any graph accepted by a TM in linear space, with waste $min\{n/2 + f(n), n\}$, where $f(n)$ is the number of faults in the execution. We finally prove that increasing the permissible waste to $min\{2n/3 + f(n),\ n\}$ allows the construction of graphs accepted by an $O(n^2)$-space Turing Machine, which is asymptotically the maximum simulation space that we can hope for in this model.

In the full version, we also provide a protocol Π' based on restarts such that, given any network constructor Π with notifications, Π' is a fault-tolerant version of Π without waste. However, the required memory per node in this protocol is $O(\log n)$ bits.

Finally, in Section 5 we conclude and discuss further research directions opened by this work.

The following table summarizes all results proved in this paper.

Constructible languages		
Without notifications		With notifications
Unbounded faults	Bounded faults	Unbounded faults
Only Spanning Clique	Non-hereditary impossibility	Fault-tolerant protocols: Spanning Star, Cycle Cover, Spanning Line
Strong impossibility even with linear waste	A representation of any finite graph (partial constructibility)	Universal Fault-tolerant Constructors (with waste)
	Any constructible graph language with linear waste	Universal Fault-tolerant Restart (without waste)

Table 1: Summary of our results.

Due to space constraints, several technical details are omitted from this extended abstract. A full version with all proofs can be found at [MST19].

2 Model and Definitions

A Network Constructor (NET) is a distributed protocol defined by a 4-tuple $(Q, q_0, Q_{out}, \delta)$, where Q is a finite set of node-states, $q_0 \in Q$ is the initial node-state, $Q_{out} \subseteq Q$ is the set of output node-states, and $\delta : Q \times Q \times \{0,1\} \to Q \times Q \times \{0,1\}$ is the transition function, where $\{0,1\}$ is the set of edge states.

In the generic case, there is an *underlying interaction graph* $G_U = (V_U, E_U)$ specifying the permissible interactions between the nodes, and on top of G_U, there is a dynamic overlay graph $G_O = (V_O, E_O)$. A mapping function F maps every node in the overlay graph to a distinct underlay node. In this work, G_U is a *complete undirected interaction graph*, i.e., $E_U = \{uv : u, v \in V_U \text{ and } u \neq v\}$, while the overlay graph consists of a population of n initially *isolated* nodes (also called *agents*).

The NET protocol is stored in each node of the overlay network, thus, each node $u \in G_O$ is defined by a state $q \in Q$. Additionally, each edge $e \in E_O$ is defined by a binary state (*active/connected* or *inactive/disconnected*). Initially, all nodes are in the same state q_0 and all edges are inactive. The goal is for the nodes, after interacting and activating/deactivating edges for a while, to end up with a desired stable overlay graph, which belongs to some graph language L.

During a (pairwise) interaction, the nodes are allowed to access the state of their joining edge and either activate it (state = 1) or deactivate it (state = 0). When the edge state between two nodes $u, v \in G_O$ is activated, we say that u and v are *connected*, or *adjacent* at that time t, and we write $u \underset{t}{\sim} v$.

In this work, we present a version of this model that allows *adversarial crash failures*. A crash (or *halting*) failure causes an agent to cease functioning and play no further role in the execution. This means that all the adjacent edges of $F(u) \in G_U$ are removed from E_U, and, at the same time, all the adjacent edges of $u \in G_O$ become inactive.

The execution of a protocol proceeds in discrete steps. In every step, an edge $e \in E_U$ between two nodes $F(u)$ and $F(v)$ is selected by an *adversary scheduler*, subject to some *fairness* guarantee. The corresponding nodes u and v interact with each other and update their states and the state of the edge $uv \in G_O$ between them, according to a joint transition function δ. If two nodes in states q_u and q_v with the edge joining them in state q_{uv} encounter each other, they can change into states q'_u, q'_v and q'_{uv}, where $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$. In the original model, G_U is the complete directed graph, which means that during an interaction, the interacting nodes have distinct roles. In our protocols, we consider a more restricted version, that is, *symmetric* transition functions, as we try to keep the model as minimal as possible. In particular, $\delta(q_u, q_v, q_{uv}) = (a, b, c)$ implies $\delta(q_v, q_u, q_{vu}) = (b, a, c)$.

A configuration is a mapping $C : V_I \cup E_I \to Q \cup \{0, 1\}$ specifying the state of each node and each edge of the interaction graph. An execution of the protocol on input I is a finite or infinite sequence of configurations, $C_0, C_1, C_2, \ldots$, each of which is a set of states drawn from $Q \cup \{0, 1\}$. In the initial configuration C_0, all nodes are in state q_0 and all edges are inactive. Let q_u and q_v be the states of the nodes u and v, and q_{uv} denote the state of the edge joining them. A configuration C_k is obtained from C_{k-1} by one of the following types of transitions:

1. **Ordinary transition:** $C_k = (C_{k-1} - \{q_u, q_v, q_{uv}\}) \cup \{q'_u, q'_v, q'_{uv}\}$ where $\{q_u, q_v, q_{uv}\} \subseteq C_{k-1}$ and $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$.
2. **Crash failure:** $C_k = C_{k-1} - \{q_u\} - \{q_{uv} : uv \in E_I\}$ where $\{q_u, q_{uv}\} \subseteq C_{k-1}$.

We say that C' is *reachable from* C and write $C \rightsquigarrow C'$, if there is a sequence of configurations $C = C_0, C_1, \ldots, C_t = C'$, such that $C_i \to C_{i+1}$ for all i, $0 \leqslant i < t$. The fairness condition that we impose on the scheduler

is quite simple to state. Essentially, we do not allow the scheduler to 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. Equivalently, we require that any configuration that is always reachable is eventually reached.

We define the output of a configuration C as the graph $G(C) = (V, E)$ where $V = \{u \in V_O : C(u) \in Q_{out}\}$ and $E = \{uv : u, v \in V,\ u \neq v,\ \text{and } C(uv) = 1\}$. If there exists some step $t \geq 0$ such that $G(C_i) = G$ for all $i \geq t$, we say that the output of an execution $C_0, C_1, \ldots$ *stabilizes* (or *converges*) to graph G, every configuration C_i, for $i \geq t$, is called *output-stable*, and t is called the *running time* under our scheduler. We say that a protocol Π stabilizes eventually to a graph G of *type* L if and only if after a finite number of pairwise interactions, the graph defined by 'on' edges does not change and belongs to the graph language L.

Definition 1. *We say that a protocol Π* constructs *a graph language L if: (i) every execution of Π on n nodes stabilizes to a graph $G \in L$ s.t. $|V(G)| = n$ and (ii) $\forall G \in L$ there is an execution of Π on $|V(G)|$ nodes that stabilizes to G.*

Definition 2. *We say that a protocol Π* partially constructs *a graph language L, if: (i) requirement (i) from Definition 1 holds and (ii) $\exists G \in L$ s.t. no execution of Π on $|V(G)|$ nodes stabilizes to G.*

Definition 3 (Fault-tolerant protocol). *Let Π be a NET protocol that, in a failure-free setting, constructs a graph $G \in L$. Π is called* f-fault-tolerant *if for any population size $n > f$, any execution of Π constructs a graph $G \in L$, where $|V(G)| = n - f$. We also call Π* fault-tolerant *if the same holds for any number $f \leq n - 2$ of faults.*

Definition 4 (Constructible language). *A graph language L is called* constructible *(*partially constructible*) if there is a protocol that constructs (partially constructs) it. Similarly, we call L* constructible under f faults*, if there is an f-fault-tolerant protocol that constructs L, where f is an upper bound on the maximum number of faults during an execution.*

Definition 5 (Critical node). *Let G be a graph that belongs to a graph language L. Call u a* critical node *of G if by removing u and all its edges, the resulting graph $G' = G - \{u\} - \{uv : v \sim u\}$, does not belong to L. In other words, if there are no critical nodes in G, then any (induced) subgraph G' of G that can be obtained by removing nodes and all their edges, also belongs to L.*

Definition 6 (Hereditary Language). *A graph language L is called* hereditary *if for any graph $G \in L$, every induced subgraph of G also belongs to L. In other words, there is no graph $G \in L$ with critical nodes.*

This notion is known in the literature as *hereditary property* of a graph w.r.t. (with respect to) some graph language L. Observe that if there exists a graph G s.t. for any induced subgraph G' of G, $G' \in L$, does not imply that the same holds for any graph in L. Some examples of hereditary languages are "Bipartite graph", "Planar graph", "Forest of trees", "Clique", "Set of cliques", and "Maximum node degree $\leq \Delta$".

In this work, unless otherwise stated, a graph language L is an infinite set of graphs satisfying the following properties:

1. (*No gaps*): For all $n \geq$ c, where $c \geq 2$ is a finite integer, $\exists G \in L$ of order n.
2. (*No Isolated Nodes*): $\forall G \in L$ and $\forall u \in V(G)$, it holds that $d(u) \geq 1$ (where $d(u)$ is the degree of u).

Even though graph languages are not allowed to contain isolated nodes, there are cases in which a protocol might be allowed to output one or more isolated nodes. In particular, if a protocol Π constructing L is allowed a waste of at most w, then whenever Π is executed on n nodes, it must output a graph $G \in L$ of order $|V(G)| \geq n - w$, leaving at most w nodes in one or more separate components (could be all isolated).

3 Network Constructors without Fault Notifications

In this section, we study the constructive power of the original NET model in the presence of bounded and unbounded crash faults when no form of notification is available to the nodes.

3.1 Unbounded Number of Faults

We here consider the setting where the number of faults can be any number up to $n-2$. We prove that the only constructible graph language is *Spanning Clique* $= \{G : G \text{ is a spanning clique}\}$.

We first present a protocol which constructs the language *Spanning Clique* and we show that it can tolerate any number of faults. Let *Clique* be the following 2-state *symmetric* protocol.

Protocol Clique: $Q = \{b, r\}$, initial state b, and transition function $\delta : (b,\ b,\ 0) \to (b,\ r,\ 0)$, $(b,\ r,\ 0) \to (r,\ r,\ 0)$, $(r,\ r,\ 0) \to (r,\ r,\ 1)$

Lemma 1. Clique *is a fault-tolerant protocol for* Spanning Clique.

In addition, we show that (due to the power of the adversary), no other graph language is constructible under unbounded faults.

Lemma 2. *Let Π be a protocol constructing a language L and $G \in L$ be a graph that Π outputs on $|V(G)|$ nodes. If G has an independent set $S \subseteq V$, s.t. $|S| \geq 2$, then there is an execution of Π on n nodes which stabilizes on $|S|$ isolated nodes (where $|S| = n - f$ and f is the number of faults in that execution).*

Theorem 1. *Let L be any graph language such that $L \neq$ Spanning Clique. Then, there is no protocol that constructs L if an unbounded number of crash failures may occur.*

The proof following theorem is a direct application of Lemma 2.

Theorem 2. *Let L be any graph language such that the graphs $G \in L$ have maximum independent sets whose size grows with $|V(G)|$. If the useful space of protocols is required to grow with n, then there is no protocol that constructs L in the unbounded-faults case.*

3.2 Bounded Number of Faults

The exact characterization established above, shows that under unbounded failures and without further assumptions, we cannot hope for non-trivial constructions. We now relax the power of the faults adversary, so that there is a *finite upper bound* f on the number of faults. In particular, fixing any such $0 \leq f \leq n$ in advance, it is guaranteed that $\forall n \geq 0$ and all executions of a protocol on n nodes, at most f nodes may fail during the execution. Then the class of constructible graph languages is naturally parameterized in f.

We first show that non-hereditary languages are not constructible under a single fault.

Theorem 3. *If there exists a critical node in G, there is no 1-fault-tolerant NET protocol that stabilizes to it.*

By Definition 6 and Theorem 3 it follows that.

Corollary 1. *If a graph language L is non-hereditary, it is impossible to be constructed under a single fault.*

Note that this does not imply that any hereditary language is constructible under a constant number of faults. We leave this as an interesting open problem.

On the positive side we show that in the case of a bounded number of faults, there is a non-trivial class of languages that is partially constructible. Consider the class of graph languages defined as follows. Any such language $L_{D,f}$ in the family is uniquely specified by a graph $D = ([k], H)$ and the finite upper bound $f < k$ on the number of faults. A graph $G = (V, E)$ belongs to $L_{D,f}$ iff there are k partitions $V_1, V_2, \ldots, V_k$ of V s.t. for all $1 \leq i, j \leq k$, $||V_i| - |V_j|| \leq f + 1$. In addition, E is constructed as follows. The graph $D = ([k], H)$, possibly containing self-loops, defines a neighboring relation between the k partitions. For every $(i, j) \in H$ (where possibly $i = j$), E contains all edges between partitions V_i and V_j, i.e., a complete bipartite graph between them (or a clique in case $i = j$). As no isolated nodes are allowed, every V_i must be fully connected to at least one V_j (possibly itself).

We first consider the case where $k = 2^\delta$, for some constant $\delta \geq 0$, and we provide a protocol that divides the population into k partitions. The protocol works as follows: initially, all nodes are in state c_0 (we call this the partition 0). When two nodes in states c_i, where $i \geq 0$ interact with each other, they update their states to c_{2i+1} and c_{2i+2}, moving to partitions $2i + 1$ and $2i + 2$ respectively. Interactions between nodes in different c-states (c_i, c_j, where $i \neq j$) do not affect the configuration. When $j = 2i + 1 \geq k - 1$ (or $j = 2i + 2 \geq k - 1$) for the first time, it means that the node has reached its final partition. It updates its state to P_m, where $m = j - k + 1$, thus, the final partitions are $\{P_0, P_1, \ldots, P_{k-1}\}$.

This process divides each partition into two partitions of equal size. However, in the case where the number of nodes is odd, a single node remains unmatched. For this reason, all nodes participate to the final formation of H regardless of whether they have reached their final partitions or not. There is a straightforward mapping of each internal partition to a distinct leaf of the binary tree, that is, each partition c_i behaves as if it were in partition P_i. In order to avoid false connections between the partitions, we also allow the nodes to disconnect from each other if they move to a different partition. This process guarantees that eventually all nodes end up in a single partition, and their connections are strictly described by H.

Lemma 3. *In the absence of faults, the above protocol divides the population into k partitions of at least $n/k - 1$ nodes each.*

Lemma 4. *In the case where up to f faults occur during the execution, each final partition has at least $n/k - f - 1$ nodes, where k is the number of partitions and $f < k$.*

Corollary 2. $||V_i| - |V_j|| \leq f + 1$, $\forall 1 \leq i, j \leq k$.

Theorem 4. *The language $L_{D,f}$, where k is a constant number, is partially constructible under f faults.*

Finally, we show that if we permit a waste linear in n, any graph language that is constructible in the fault-free NET model, becomes constructible under a bounded number of faults.

Theorem 5. *Take any NET protocol Π of the original fault-free model. There is a NET Π' such that when at most f faults may occur on any population of size n, Π' successfully simulates an execution of Π on at least $\frac{n}{2f} - f$ nodes.*

4 Notified Network Constructors

In light of the impossibility results of Section 3, we allow fault notifications when nodes crash, aiming at constructing a larger class of graph languages. In particular, we introduce a *fault flag* in each node, which is initially zero. When a node u crashes at time t, every node v which was adjacent to u at time t is notified, that is, the fault flag of all v becomes 1. In the case where u is an isolated node (i.e., it has no active edges), an arbitrary node w in the graph is notified, and its fault flag becomes 2. Then, the fault flag becomes immediately zero after applying a corresponding rule from the transition function.

More formally, the set of node-states is $Q \times \{0, 1, 2\}$, and for clarity in our descriptions and protocols, we define two types of transition functions. The first one determines the node and connection state updates of pairwise interactions ($\delta_1 : Q \times Q \times \{0, 1\} \to Q \times Q \times \{0, 1\}$), while the second transition function determines the node state updates due to fault notifications ($\delta_2 : Q \times \{0, 1, 2\} \to Q \times \{0\}$). This means that during a step t that a node u crashes, all its adjacent nodes are allowed to update their states based on δ_2 at that same step. If there are no any adjacent nodes to u, an arbitrary node is notified, thus, updating its state based on δ_2 at step t.

Proposition 1. *We provide fault-tolerant protocols for* spanning star *and* cycle cover *(see Protocol* 3 *and Protocol* 4 *in [MST19]).*

4.1 Universal Fault-Tolerant Constructors

In this section, we ask whether there is a generic fault-tolerant constructor capable of constructing a large class of graphs. We first give a fault-tolerant protocol that constructs a spanning line, and then we show that we can simulate a given TM on that line, tolerating any number of crash faults. The rules of the protocol and the proof of its correctness can be found in [MST19]. Finally, we exploit that in order to construct any graph language that can be decided by an $O(n^2)-$space TM, paying at most linear waste.

Lemma 5. *FT Spanning Line (see Protocol* 5 *in [MST19]) is fault-tolerant.*

Lemma 6. *There is a NET Π (with notifications) such that when Π is executed on n nodes and at most k faults can occur, where $0 \leq k < n$, Π will eventually simulate a given TM M of space $O(n-k)$ in a fault-tolerant way.*

Lemma 7. *There is a fault-tolerant NET Π (with notifications) which partitions the nodes into two groups U and D with waste at most $2f(n)$, where $f(n)$ is an upper bound on the number of faults that can occur. U is a spanning line with a unique leader in one endpoint and can eventually simulate a TM M. In addition, there is a perfect matching between U and D.*

Theorem 6. *For any graph language L that can be decided by a linear space TM, there is a fault-tolerant NET Π (with notifications) that constructs a graph in L with waste at most $min\{n/2+f(n),\ n\}$, where $f(n)$ is an upper bound on the number of faults that can occur.*

We now show that if the constructed network is required to occupy $1/3$ instead of half of the nodes, then the available space of the TM-constructor dramatically increases from $O(n)$ to $O(n^2)$. We provide a protocol which partitions the population into three sets U, D and M of equal size $k = n/3$ (see Protocol 6 in [MST19]). The idea is to use the set M as a $\Theta(n^2)$ binary memory for the TM, where the information is stored in the $k(k-1)/2$ edges of M.

Theorem 7. *For any graph language L that can be decided by an $O(n^2)-$space TM, there is a protocol that constructs L equiprobably with waste at most $min\{2n/3+f(n),\ n\}$, where $f(n)$ is an upper bound on the number of faults.*

5 Conclusions and Open Problems

A number of interesting problems are left open for future work. Our only exact characterization was achieved in the case of unbounded faults and no notifications. If faults are bounded, non-hereditary languages were proved impossible to construct without notifications but we do not know whether hereditary languages are constructible. Relaxations, such as permitting waste or partial constructibility were shown to enable otherwise impossible transformations, but there is still work to be done to completely characterize these cases. In case of notifications, we managed to obtain fault-tolerant universal constructors, but it is not yet clear whether the assumptions of waste and local coin tossing that we employed are necessary and how they could be dropped. Apart from these immediate technical open problems, some more general related directions are the examination of different types of faults such as random, Byzantine, and communication/edge faults. Finally, a major open front is the examination of fault-tolerant protocols for stable dynamic networks in models stronger than NETs.

References

AAD+06. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18[4]:235–253, March 2006.

AAER07. D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20[4]:279–304, November 2007.

AAFJ08. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3[4]:1–28, 2008.

BBB13. J. Beauquier, P. Blanchard, and J. Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *International Conference on Principles of Distributed Systems*, pages 38–52. Springer, 2013.

CLV+17. C. Cooper, A. Lamani, G. Viglietta, M. Yamashita, and Y. Yamauchi. Constructing self-stabilizing oscillators in population protocols. *Information and Computation*, 255:336–351, 2017.

DDG+14. Z. Derakhshandeh, S. Dolev, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann. Brief announcement: amoebot–a new model for programmable matter. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 220–222. ACM, 2014.

DDG+18. J. J. Daymude, Z. Derakhshandeh, R. Gmyr, A. Porter, A. W. Richa, C. Scheideler, and T. Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17[1]:81–96, 2018.

DGFGR06. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. When birds die: Making population protocols fault-tolerant. In *IEEE 2nd Intl*

Conference on Distributed Computing in Sensor Systems (DCOSS), volume 4026 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, June 2006.

DIM93. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7[1]:3–16, Nov 1993.

DLFI+17. G. A. Di Luna, P. Flocchini, T. Izumi, T. Izumi, N. Santoro, and G. Viglietta. Population protocols with faulty interactions: the impact of a leader. In *International Conference on Algorithms and Complexity (CIAC)*, pages 454–466. Springer, 2017.

DLFS+19. G. A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. Shape formation by programmable particles. *Distributed Computing*, pages 1–33, 2019.

Dol00. S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

DT01. B. Ducourthial and S. Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14[3]:147–162, Jul 2001.

GK10. N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70[4]:406 – 415, 2010.

GKR10. K. Gilpin, A. Knaian, and D. Rus. Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2485–2492. IEEE, 2010.

GR09. R. Guerraoui and E. Ruppert. Names trump malice: Tiny mobile agents can tolerate byzantine failures. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 484–495. Springer, 2009.

MCS11. O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Mediated population protocols. *Theoretical Computer Science*, 412[22]:2434–2450, May 2011.

Mic18. O. Michail. Terminating distributed construction of shapes and patterns in a fair solution of automata. *Distributed Computing*, 31[5]:343–365, 2018.

MOKY12. R. Mizoguchi, H. Ono, S. Kijima, and M. Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25[6]:451–460, 2012.

MS16. O. Michail and P. G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29[3]:207–237, 2016.

MS17. O. Michail and P. G. Spirakis. Network constructors: A model for programmable matter. In B. Steffen, C. Baier, M. van den Brand, J. Eder, M. Hinchey, and T. Margaria, editors, *SOFSEM 2017: Theory and Practice of Computer Science*, pages 15–34, Cham, 2017. Springer International Publishing.

MSS19. O. Michail, G. Skretas, and P. G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

MST19. O. Michail, P. G. Spirakis, and M. Theofilatos. Fault tolerant network constructors. *arXiv preprint arXiv:1903.05992*, 2019.

Pel09. D. Peleg. As good as it gets: Competitive fault tolerance in network structures. In R. Guerraoui and F. Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 35–46, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.