Fault Tolerant Network Constructors[⋆,⋆⋆]

Othon Michail[∗], Paul G. Spirakis, Michail Theofilatos

Department of Computer Science, University of Liverpool, Liverpool, 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. In particular, if an unbounded number of crash faults may occur, we prove that (i) the only constructible graph language is that of spanning cliques and (ii) a strong impossibility result holds even if the size of the graphs that the protocol outputs in populations of size n needs to grow with n (the remaining nodes being *waste*). 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*. Our main result here is a *fault-tolerant universal constructor*: We develop a fault-tolerant protocol for *spanning line* and use it to simulate a linear-space Turing Machine M. This allows a fault-tolerant construction of any graph accepted by M in linear space, with waste $min\{n/2 + f(n),\ n\}$, where $f(n)$ is the number of faults in the execution. We then 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. 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, population, 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 links when they meet. This model, called *Network Constructors*, was introduced in [23], and is based on the *Population Protocol* (PP) model [1, 2] and the *Mediated Population Protocol* (MPP) model [21]. We are interested in answering questions like the following: If one or more faults can affect the formation process, can we always re-stabilize to a correct graph, and if not, what is the class of graph languages for which

[⋆]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.

[⋆⋆]A preliminary version of the results in this paper has appeared in [25].

[∗]Corresponding author. Postal Address: Department of Computer Science, University of Liverpool, Ashton Street, Liverpool L69 3BX, UK.

Email addresses: `Othon.Michail@liverpool.ac.uk` (Othon Michail), `P.Spirakis@liverpool.ac.uk` (Paul G. Spirakis), `theofilatos.michail@gmail.com` (Michail Theofilatos)

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 [20]) is a theoretical model that may be viewed as a minimal model for programmable matter operating in a dynamic environment [24]. 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 [17], to smart materials, and many theoretical models (see, e.g., [10, 8, 22, 12] 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 [3, 4, 7] and MPPs [26] do not apply here.

In [23], 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. In particular, if Π is a protocol that constructs a graph language L, then a crash fault may result to a configuration C such that no execution of Π starting from C stabilizes to a graph in L. This means that when faults occur, the population must perform some computations so as to reach a configuration where all executions of Π will again stabilize to a graph in L. The problem is then to study self-stabilizing protocols under crash faults. 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. The cases of sequential and parallel occurrence of faults are equivalent to each other 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 [14, 13], 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 [18, 15] 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.* [3] 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.* [9] studied the issue of correctly computing functions on the node inputs in the Population Protocol model [1], 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. In the context of fault tolerance, [11] uses a leader to make any protocol tolerant to omission failures (i.e., failure by an agent to read its partner's state during an interaction). In [16], Fischer and Jiang introduced the $\Omega?$ detectors in order to solve leader election under crash or transient faults. An eventual leader detector $\Omega?$ is an oracle that eventually detects the presence or absence of a leader in the population.

Guerraoui and Ruppert [19] 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 [27] 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 [23], 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 needs to grow with n (i.e., $\omega(1)$), and 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*. This is essentially a failure detector ([6, 5]) that provides information about crash fault events in some nodes of the network. In [6] the failure detector $\Diamond S$ eventually outputs a set of crashed process identities at each process of the network. In our work, 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 4.2). 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 [23], where they show several universality

Constructible languages		
Without notifications		With notifications
Unbounded faults (Section 3.1)	Bounded faults (Section 3.2)	Unbounded faults
Only Spanning Clique	Non-hereditary impossibility	Fault-tolerant protocols: Spanning Star, Cycle Cover, Spanning Line (Section 4.1)
Strong impossibility even with linear waste	A representation of any finite graph (partial constructibility)	Universal Fault-tolerant Constructors (with waste) (Section 4.2)
	Any constructible graph language with linear waste	Universal Fault-tolerant Restart (without waste) (Section 4.3)

Table 1: Summary of our results.

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 the input 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 order to give fault-tolerant protocols without waste, we design a protocol that can be composed in parallel with any protocol in order to make it fault-tolerant. The idea is to restart the protocol whenever a crash failure occurs. We show that restarting is impossible with constant local memory, if the nodes may form a linear (in the population size) number of connections; hence, to overcome this we supply the agents with logarithmic memory (Section 4.3).

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. We show that restarting is impossible with constant local memory, if the nodes may form a linear (in the population size) number of connections; hence, 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.

Table 1 summarizes all results proved in this paper.

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 $F(u) \in G_U$, where $F(u)$ is the node of the underlying graph that u is mapped to, 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 the following constraint that is imposed by the fact that the edges of the interaction graph are undirected. In particular, $\delta(q_u, q_v, q_{uv}) = (a, b, c)$ implies $\delta(q_v, q_u, q_{vu}) = (b, a, c)$, for any $q_u, q_v \in Q$.

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 $G_O(C_i) = G$ 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.

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).

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 *any execution of* Π *constructs a graph* $G \in L$*, where* $|V(G)| \geq n - f$*, and* $f < n$ *is an upper bound on the number of faults. We also call* Π fault-tolerant *if the same holds for any number* $f \leq n-2$ *of faults.*

Definition 4 (Waste and useful space)**.** *We say that a protocol* Π constructs *a graph language* L *with* waste w *if: (i) every execution of* Π *on* n *nodes stabilizes to a graph* $G \in L$ *s.t.* $n - w \leq |V(G)| \leq n$ *and (ii)* $\forall G \in L$ *there is an execution of* Π *on* n *nodes s.t.* $|V(G)| \leq n \leq |V(G)| + w$ *that stabilizes to* G*. This implies that the waste includes all crashed nodes and any auxiliary nodes required by* Π *to construct* G*. Finally, we call* $|V(G)|$ *the* useful space*.*

Definition 5 (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 6 (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 7 (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 $\leq \Delta$”.

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. We start from the case in which the number of nodes that crash during an execution can be anything from 0 up to $n-2$ nodes. We are interested in characterizing the class of constructible graph languages. Observe that we cannot trivially conclude that the adversary can always leave us with just 2 nodes, only allowing our protocols to form a line of length 1. This is because our definition of constructible languages under faults takes into account all possible executions with f faults, for all values of $f \in \{0, 1, \ldots, n-2\}$. We show that in the case where the number of faults cannot be bounded by a constant number, the only language that is constructible is the $L_c = \{G : G \text{ is a spanning clique}\}$.

We then consider the setting where only a constant number of faults are allowed, and we show that no language L is constructible under a single fault, if L is not Hereditary. However, if we allow linear waste in the population, any language that is constructible without faults, becomes constructible under a constant number of faults.

Finally, we show a family of graph languages that is partially constructible (without waste in the population). The exact characterization of the class of partially constructible languages remains as an open problem.

3.1. Unbounded Number of Faults

We consider here 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* $= \{G : G$ is a spanning clique$\}$.

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

Protocol 1 Clique

$Q = \{b\}$
Initial state: b

δ :
$(b,\ b,\ 0) \rightarrow (b,\ b,\ 1)$

\\\\All transitions that do not appear have no effect.

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

Clearly, for any number $f < n$ of faults, where n is the population size, Protocol 1 constructs the language *Spanning Clique*.

By Lemma 1, we know that the language *Spanning Clique* is constructible under $n-2$ faults. To clarify, this means that for any execution of Protocol 1 on n nodes, f of which crash ($f \in \{0, 1, \ldots, n-2\}$), Protocol 1 is guaranteed to stabilize to a clique of order $n-f$.

We will now prove 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).*

Proof. Consider an execution of Π that outputs G. By definition, there is a point in this execution after which no further edge updates can occur (no matter what the infinite execution suffix will be). Take any configuration C_{stable} after that point and consider its sub-configuration C_S induced by the independent set S. Observe that C_S encodes the state of each node $u \in S$ in that particular stable configuration C_{stable}. Denote also by Q_S the multiset of all states assigned by C_S to the nodes in S.

Every state in Q_S is reachable (in the sense that there exists an execution that produces it). For each $q \in Q_S$ consider the smallest population V_q in which there is some execution a_q of protocol Π that produces state q. Consider the population $V = \bigcup_{q \in Q_S} V_q$ (or equivalently of size $n = \sum_{q \in Q_S} |V_q|$).

For each V_q in population V we execute a_q until q is produced on some node u_q. After this, every $q \in Q_S$ is present in the population V. Then, the adversary crashes all nodes in $V_q \setminus \{u_q\}$ (i.e., only u_q remains alive in each V_q). This leaves the execution with a set of alive nodes equivalent in cardinality and configurations to the independent set S under C_S.

The above construction is a finite prefix of fair executions. For the sake of contradiction, assume that in any fair continuation of the above prefix, Π eventually stabilizes to a graph with no isolated nodes (as required by the fact that Π constructs a graph language L). Take one such continuation γ. As γ starts from a configuration in all respects equivalent to that of S under C_{stable}, it follows that γ can also be applied to C_{stable} and in particular on the independent set S starting from C_S. It follows that γ must have exactly the same effect as before, that is, eventually it will cause the activation of at least one edge between the nodes in S. But this violates the fact that C_{stable} is a stable configuration, therefore no edge could have been activated by Π in the continuation, implying that the continuation must have been an execution stabilizing on $|S|$ isolated nodes. $\square$

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.*

Proof. As $L \neq$ *Spanning Clique*, there exists $G \in L$ such that G is not complete (and by definition no $G' \in L$ has isolated nodes). Therefore G has an independent set S of size at least 2. If there exists a protocol Π that constructs L, then by Lemma 2 there must be an execution of Π which stabilizes on at least 2 isolated nodes. The latter is a stable output not in L, therefore a contradiction. □

Theorem 2. *If an unbounded number of faults may occur, the Spanning Clique is the only constructible language.*

Proof. Directly from Lemma 1 and Theorem 1. □

Theorem 3. *Let L be any graph language such that the graphs $G \in L$ have maximum independent sets whose size grows with $|V(G)|$ (i.e., $\omega(1)$). 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.*

Proof. The proof is a direct application of Lemma 2. As the size of the maximum independent set of G grows with $|V(G)|$ in L, and the useful space is a non-constant function of n, it follows that, as n grows, the stable output-graph (on the useful space) has an independent set of size that grows with n (consider, for example, the leaves of binary trees of growing size as such a growing independent set). As any such stable independent set of size $g(n)$ implies that another execution has to stabilize to $g(n)$ isolated nodes, it follows that any protocol for L would produce infinitely many stable outputs of isolated nodes. The latter is contradicting the fact that the protocol constructs L. □

3.2. Bounded Number of Faults

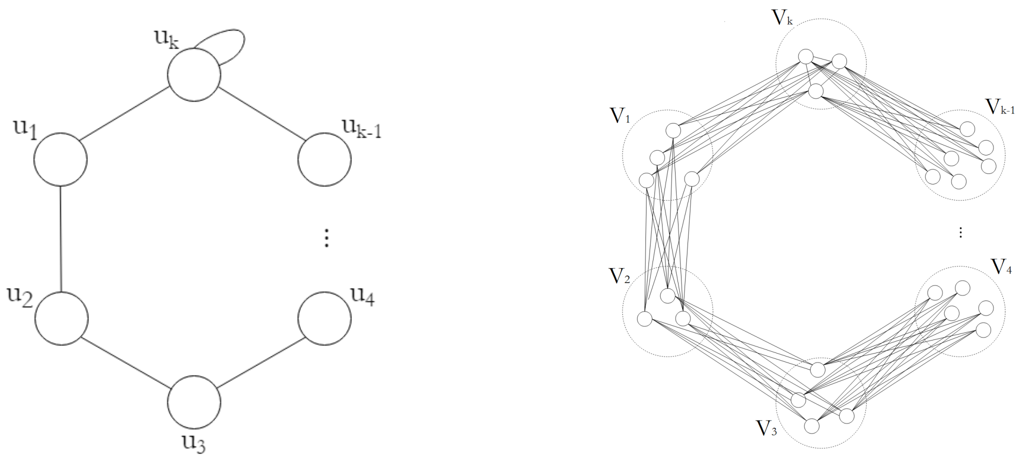
The exact characterization established above, shows that under unbounded failures and without further assumptions, we cannot hope for non-trivial constructions. We now relax the power of the faults adversary, so that there is a *finite upper bound* f on the number of faults. In particular, for any $n \geq 1$, and fixing any such $0 \leq f \leq n$ in advance, it is guaranteed that for 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 4. *If there exists a critical node in G, there is no 1-fault-tolerant NET protocol that stabilizes to it.*

Proof. Let Π be a NET protocol that constructs a graph language L, tolerating one crash failure. Consider an execution $\mathcal{E}$ and a sequence of configurations $C_0, C_1, \ldots$ of $\mathcal{E}$. Assume a time t that the output of $\mathcal{E}$ has stabilized to a graph $G \in L$ (i.e., $G(C_i) = G$, $\forall i \geq t$). Let u be a critical node in G. Assume that the scheduler removes u and all its edges (crash failure) at time $t' > t$, resulting to a graph $G' \notin L$. In order to fix the graph (i.e., re-stabilize to a graph $G'' \in L$), the protocol must change at some point t'' the configuration. This can only be the result of a state update on some node v. Now, call $\mathcal{E}'$ the execution that node u does not crash and, besides that, is the same as $\mathcal{E}$. Then, between t' and t'' the node v has the same interactions as in the previous case where node u crashed. This results to the same state update in v, since it cannot distinguish $\mathcal{E}$ from $\mathcal{E}'$. The fact that u either crashes or not, leads to the same result (i.e., v tries to fix the graph thinking that u has crashed). This means that if we are constantly trying to detect faults in order to deal with them, this would happen indefinitely and the protocol would never be stabilizing. Consider that the network has stabilized to G. At some point, because of the infinite execution, a node will surely but wrongly detect a crash failure. Thus, G has not really stabilized. □

By Definition 7 and Theorem 4 it follows that.

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



(a) $D = ([k], H)$

(b) Graph of supernodes

Figure 1: In 1a, D defines a ring of size k. In 1b, each node of D corresponds to a set of nodes (or supernode), while for each edge of D between two nodes u_i and u_j, all nodes of V_i are connected to all nodes of V_j and vice versa.

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 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 $1 \le i, j \le k$, $||V_i| - |V_j|| \le f + 1$. 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). In Figure 1, we present an example of a graph that belongs to $L_{D,f}$, where D defines a ring graph.

We first consider the case where $k = 2^\epsilon$, for some constant $\epsilon \in \mathbb{N}_0$, 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 $i \ge 0$ 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 $i \ne j$) do not affect the configuration. When $j = 2i + 1 \ge k - 1$ (or $j = 2i + 2 \ge k - 1$) 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, Protocol 2, divides the population into k partitions of at least $n/k - 1$ nodes each.*

Proof. Initially all nodes are in state c_0. When two c_0 nodes interact with each other, one of them becomes c_1 and the other one c_2. This means that all n nodes split into two partitions of equal size. No node can

Protocol 2 Graph of Supernodes

$Q = \{c_i, P_j\} \times \{0,1\}$, $0 \le i \le 2(k-1)$, $0 \le j \le k-1$
Initial state: c_0

δ :
\\Partitioning
1. $(c_i,\ c_i,\ 0) \to (c_{2i+1},\ c_{2i+2},\ 0)$, if $(i+1) < k$
2. $(c_i,\ \cdot,\ \cdot) \to (P_j,\ \cdot,\ \cdot)$, if $(i \ge k-1)$, $j = i-k+1$

\\Formation of graph H
3. $(P_i, P_j, 0) \to (P_i, P_j, 1)$, if $(i,j) \in H$
4. $(P_i, P_j, 1) \to (P_i, P_j, 0)$, if $(i,j) \notin H$
5. $(c_i, P_j, 0) \to (c_i, P_j, 1)$, if $(i,j) \in H$
6. $(c_i, P_j, 1) \to (c_i, P_j, 0)$, if $(i,j) \notin H$
7. $(c_i, c_j, 0) \to (c_i, c_j, 1)$, if $(i,j) \in H$
8. $(c_i, c_j, 1) \to (c_i, c_j, 0)$, if $(i,j) \notin H$

\\All transitions that do not appear have no effect.

become c_0 again at any time during the execution. In addition, there is only one partition c_j that produces nodes of some other partition c_i, where i is either $2j+1$ or $2j+2$, and the size of them are half the size of c_j. This process can be viewed as traversing a labelled binary tree, until all nodes reach to their final partition. A node in state c_i has reached its final partition when $i \ge k-1$. This process describes a subdivision of the nodes, where each partition splits into two partitions of equal size. The final partitions are $\{c_{k-1}, c_k, \ldots, c_{2k-2}\}$.

Assume now that the initial population size is n_0 (level 0 of the binary tree). If n_0 is even, the size of the following two partitions c_1 and c_2 will be $n_0/2$. If n_0 is odd, one node remains unmatched, thus, the size of c_1 and c_2 will be $n_1 = \frac{n_0-1}{2}$. In the next level of the binary tree, at most one node will remain unmatched in each partition, thus $n_2 = \frac{n_1-1}{2}$. Consequently, the size of a partition in level p can be calculated recursively, and (in the worst case) it is $n_p = \frac{n_{p-1}-1}{2}$.

$$\begin{aligned} n_p &= \frac{n_{p-1}-1}{2} = \frac{n_{p-1}}{2} - \frac{1}{2} = \frac{\frac{n_{p-2}}{2} - \frac{1}{2}}{2} - \frac{1}{2} \\ &= \frac{n_{p-2}}{4} - \frac{1}{4} - \frac{1}{2} = \cdots = \frac{n_0}{2^p} - \sum_{i=1}^{p} \frac{1}{2^i} \\ &= \frac{n_0}{2^p} - (1 - 2^{-p}) > \frac{n_0}{2^p} - 1 \end{aligned} \tag{1}$$

For $p = \log k$ levels, each partition has either $\frac{n_0}{k}$ or $\frac{n_0}{k} - 1$ nodes. □

Lemma 4. *Protocol 2, stabilizes after $\Theta(kn^2)$ expected time.*

Proof. Protocol 2 operates in phases, where each phase doubles the number of partitions. After $\log k$ phases, there exist k groups in the population and the nodes terminate.

We now study the time that each group c_i needs in order to split into two partitions. Here, for simplicity, i indicates the level of a partition c in the binary tree and m_i the number of nodes of partition c_i.

Let X be a random variable (r.v.) defined to be the number of steps until all m_i nodes move to their next partitions. Call a step a success if two nodes in c_i interact, thus, moving to their next partitions. We divide the steps of the protocol into *epochs*, where epoch j begins with the step following the jth success and ends with the step at which the $(j+1)$st success occurs. Let also the r.v. X_j, $1 \leq j \leq m_i$ be the number of steps in the jth epoch.

The probability of success during the jth epoch, for $0 \leq j \leq m_i$, is $p_j = \frac{(m_i-j)(m_i-j-1)}{n(n-1)}$ and $E[X_j] = 1/p_j$. By linearity of expectation we have

$$
\begin{aligned}
E[X] &= E[\sum_{j=0}^{m_i-2} X_j] = \sum_{j=0}^{m_i-2} E[X_j] = n(n-1)\sum_{j=0}^{m_i-2} \frac{1}{(m_i-j)(m_i-j-1)} \\
&= n(n-1)\sum_{j=2}^{m_i} \frac{1}{j(j-1)} < n(n-1)\sum_{j=2}^{m_i} \frac{1}{(j-1)^2} \\
&= n(n-1)\sum_{j=1}^{m_i-1} \frac{1}{j^2} < n(n-1)\frac{\pi^2}{6} = O(n^2)
\end{aligned}
\tag{2}
$$

The above uses the fact that $m_i \leq n$ for any $i \geq 0$.

For the lower bound, observe that the last two remaining nodes in c_i need on average $n(n-1)/2$ steps to meet each other. Thus, we conclude that $E[X] = \Theta(n^2)$.

In total, $\sum_0^{\log(k)-1} 2^i = 2^{\log k} - 1 = k-1$ partitions split, thus, the total expected time to stabilization is $\Theta(kn^2)$ steps. ☐

Lemma 5. *In the case where up to f faults occur during the execution of Protocol 2, each final partition has at least $n/k - f - 1$ nodes, where k is the number of partitions and $f < k$.*

Proof. Call $\mathcal{P}_i^p$ the set of partitions that are in the binary tree starting from a partition c_i in distance p from c_i. We now study the relation between the number of faults on some partition c_i with the size of the partitions in $\mathcal{P}_i^p$.

Consider the case where f_1 crash faults occur in some partition c_i. The nodes of each partition c_i operate independently from the rest of the population, that is, they never update their states and/or connections when they interact with nodes from a different partition. Thus, if no more faults occur, we can assume that we have a failure-free execution on $|c_i| - f_1$ nodes. By Lemma 3, after p subdivisions, each partition in $\mathcal{P}_i^p$ will have $\left\lfloor \frac{|c_i|-f_1}{2^p} \right\rfloor$ nodes. Consequently, any number of faults in a partition c_i are equally split into the partitions following c_i.

Now, consider a partition $c_j \in \mathcal{P}_i^{p_1}$, where f_2 faults occur. The number of nodes of c_j is then

$$
|c_j| - f_2 = \left\lfloor \frac{|c_i| - f_1}{2^{p_1}} \right\rfloor - f_2 \tag{3}
$$

Then, all the partitions in $\mathcal{P}_j^{p_2}$, by Lemma 3 will have at least

$$
\left\lfloor \frac{|c_j| - f_2}{2^{p_2}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{|c_i|-f_1}{2^{p_1}} \right\rfloor - f_2}{2^{p_2}} \right\rfloor \geq \frac{\frac{|c_i|-f_1}{2^{p_1}} - 1 - f_2}{2^{p_2}} - 1 > \frac{\frac{|c_i|}{2^{p_1}} - f_1 - f_2 - 1}{2^{p_2}} - 1 \tag{4}
$$

nodes. This means that if $f_1 + f_2$ faults were occurred only in c_j (and not f_1 faults in c_i), then the subsequent partitions in $\mathcal{P}_j^p$ would have less nodes. This argument can be generalized for faults in any partition.

It is then obvious that in the worst case, where up to f faults can occur, a final partition (leaf of the binary tree) will have $\frac{n}{k} - f - 1$ nodes, and this is the result of f faults in that final partition. ☐

By Lemma 5:

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

By Lemma 5 and the definition of partial constructibility (Definition 5):

Theorem 5. *The language $L_{D,f}$, where k is a constant number, is partially constructible under f faults.*

We now 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 6. *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 $\frac{n}{2f} - 1$ nodes.*

Proof. Consider any constructible language L and a protocol Π that constructs it. For any bounded number of faults f, set $k = 2^{\epsilon}$, where $2^{\epsilon-1} < f < 2^{\epsilon}$. Consider a protocol Π', which consists of the rules 1 and 2 of Protocol 2. These rules partition the population into k groups, where k is an input parameter of Π'. By Lemma 5, each group has at least $n/k - f - 1$ nodes. For 2^{ϵ} partitions, the number of nodes in each partition is at least $\frac{n}{2^{\epsilon}} - f - 1$. However, as the number of partitions is strictly more than the upper bound on the number of faults ($2^{\epsilon} > f$), there exists at least one partition that no fault has occurred. In the worst case where $f = 2^{\delta}$ for some $\delta \in \mathbb{N}$, there exists at least one partition with $\frac{n}{2f} - 1$ nodes. $\square$

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 (see Figures 2a and 2b). 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 (see Figures 2c and 2d). 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 ($\delta_2 : Q \times \{1, 2\} \to Q \times \{0\}$). 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 adjacent nodes to u, an arbitrary node is notified, thus, updating its state based on δ_2 at step t.

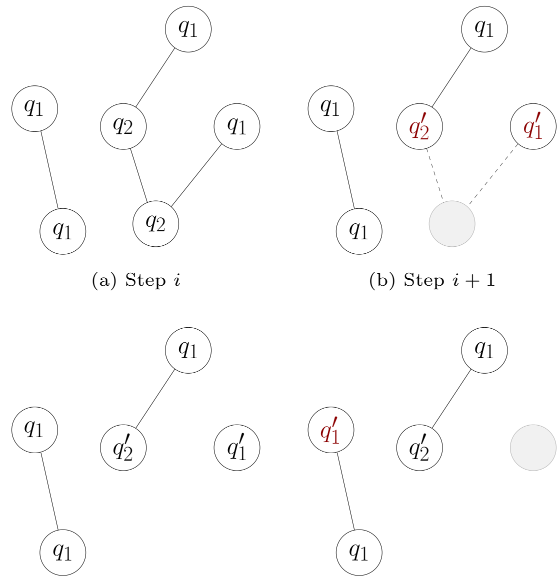
We have assumed that the faults can only occur sequentially (at most one fault per step). This assumption was equivalent to the case where many faults can occur in each step in the original NET model. However, when fault notifications are allowed, this does not hold, unless the fault flag could be used as a counter of faults in each step. We want to keep the model as minimal as possible, thus, we only allow the adversary to choose one node at most in each step to crash.

In this section, we investigate whether the additional information in each agent (the fault flag) is sufficient in order to design fault-tolerant or $f-$fault-tolerant protocols, overcoming the impossibility of certain graph languages in the NET model. Such a minimal fault notification mechanism can be exploited to construct a larger class of graph languages that in the original Network Constructors model where no form of notifications was available.

4.1. Fault-Tolerant Protocols

In this section, we give protocols for some basic network construction problems, such as *spanning star* (all $u \in G$ form a single star), *cycle cover* (set of cycles which are subgraphs of G and contain all vertices of G), and in Section 4.2 we give a fault-tolerant spanning line protocol which is part of our generic constructor capable of constructing a large class of networks.

Protocol 3 constructs a *spanning star*. Initially all nodes are in the same state b (or *black*) and they eliminate each other, becoming r (or *red*). Eventually, only one node will remain in state b which will be



(a) Step i (b) Step $i+1$

(c) Step j (d) Step $j+1$

Figure 2: An illustration of the fault notification mechanism. In the first example (Figures 2a and 2b), the gray node crashed, and the nodes that were adjacent to it at step i were notified and updated their state. At step $j > i+1$, this node along with its adjacent edges are not present. In the second example (Figures 2c and 2d), the crashed node was isolated, thus an arbitrary node was notified and updated its state.

the center of the star. In order to handle crash faults, when a *red* node is notified about a fault, it becomes *black*. In this way and because of the fact that a *red* node cannot be isolated, we guarantee that a black node will always exist in the population.

Protocol 3 FT Spanning Star

$Q = \{b,\ r\} \times \{0, 1\}$
Initial state: b

δ_1 :
\\Formation of spanning star. Eventually, only one node in state b remains.
$(b,\ b,\ 0) \to (b,\ r,\ 1)$
$(b,\ r,\ 0) \to (b,\ r,\ 1)$
$(r,\ r,\ 1) \to (b,\ b,\ 0)$
$(b,\ b,\ 1) \to (b,\ r,\ 1)$

δ_2 :
\\A leaf becomes the initial state b after a fault notification.
$(r, 1) \to (b, 0)$

Proposition 1. FT Spanning Star *(Protocol 3) is a fault-tolerant protocol that constructs a* spanning star*.*

Proof. Assume that any number of faults $f < n$ occur during an execution. Initially, all nodes are in state b (*black*). Two nodes connect with each other, if either one of them is black, or both of them are black, in which case one of them becomes r (*red*). A black node can become red only by interaction with another black node, in which case they also become connected. Thus, with no crash faults, a connected component always includes at least one black node. In addition, all isolated nodes are always in state b. This is because, if a red node removes an edge it becomes black.

Then, if a (connected) node crashes, the adjacent nodes are notified and the red nodes become black, thus, any connected component should again include at least one black node. Now, consider the case where only one black node remains in the population. Then the rest of the population (in state r) should be in the same connected component as the unique b node. Then, if b crashes, at least one black node will appear, thus, this protocol maintains the invariant, as there is always at least one black node in the population. *FT Spanning Star* then stabilizes to a star with a unique black node in the center. ☐

Protocol 4 FT Cycle-Cover

$Q = \{q_0,\ q_1,\ q_2\} \times \{0, 1\}$
Initial state: q_0

δ_1 :
$(q_0,\ q_0,\ 0) \to (q_1,\ q_1,\ 1)$
$(q_1,\ q_0,\ 0) \to (q_2,\ q_1,\ 1)$
$(q_1,\ q_1,\ 0) \to (q_2,\ q_2,\ 1)$

δ_2 :
\\The state of a node indicates its degree. A fault notification implies that the degree was decreased by one.
$(q_1, 1) \to (q_0, 0)$
$(q_2, 1) \to (q_1, 0)$

Similarly, we can show the following.

Proposition 2. FT Cycle-Cover *(Protocol 4) is a fault-tolerant protocol that forms a* cycle cover*.*

In this protocol, the state of a node indicates its degree. In particular, all nodes are initially in state q_0, indicating that they are isolated. Whenever a node in state q_i, forms a connection, it moves to state q_{i+1}. At the same time, whenever a node is notified about a fault in a neighboring node, it decreases i by one. In this protocol $0 \leq i \leq 2$, which guarantees that eventually all nodes will have degree 2, except maybe for a single node or a pair of nodes which will form a line.

4.2. 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 (i.e., a graph of size n that forms a line), and then we show that we can simulate a given TM on that line, tolerating any number of crash faults. 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 6. FT Spanning Line *(Protocol 5) is a fault-tolerant protocol that constructs a* spanning line*.*

Proof. Initially, all nodes are in state q_0 and they start connecting with each other in order to form lines that eventually merge into one.

When two q_0 nodes become connected, one of them becomes a leader (state l_0) and starts connecting with q_0 nodes (expands). A leader state l_0 is always an endpoint. The other endpoint is in state e_i (initially e_1), while the inner nodes are in state q_2. Our goal is to have only one leader l_0 on one endpoint, because l_0 are also used in order to merge lines. Otherwise, if there are two l_0 endpoints, the line could form a cycle. When two l_0 leaders meet, they connect (line merge) and a w node appears. This process corresponds to the rules 1, 2, and 3 of Protocol 5 (depicted also in Figure 3).

The w state performs a random walk on the line and its purpose is to meet both endpoints (at least once) before becoming an l_0 leader. After interacting with the first endpoint, it becomes w_1 and changes the endpoint to e_1. Whenever it interacts with the same endpoint they just swap their states from e_1, w_1

Protocol 5 FT Spanning Line

$Q = \{q_0,\ q_2,\ e_1,\ e_2,\ l_0,\ l_1,\ w,\ w_1,\ w_2\} \times \{0, 1\}$
Initial state: q_0

δ_1 :
\\Formation of lines, and merging between them.
1. $(q_0,\ q_0,\ 0) \to (e_1,\ l_0,\ 1)$
2. $(l_0,\ q_0,\ 0) \to (q_2,\ l_0,\ 1)$
3. $(l_0,\ l_0,\ 0) \to (q_2,\ w,\ 1)$

\\w nodes perform a random walk on their line.
4. $(w_i,\ q_2,\ 1) \to (q_2,\ w_i,\ 1)$
5. $(w,\ q_2,\ 1) \to (q_2,\ w,\ 1)$

\\Nodes in state w introduce a unique endpoint in state l_0 on their line.
6. $(w,\ e_i,\ 1) \to (w_i,\ e_i,\ 1)$
7. $(w_i,\ e_i,\ 1) \to (w_j,\ e_j,\ 1),\ i \neq j$
8. $(w_i,\ e_j,\ 1) \to (q_2,\ l_0,\ 1),\ i \neq j$
9. $(w,\ l_i,\ 1) \to (w_1,\ e_1,\ 1)$
10. $(w_i,\ l_i,\ 1) \to (q_2,\ l_0,\ 1)$

\\w nodes eliminate each other, until only one survives.
11. $(w_i,\ w_j,\ 1) \to (w,\ q_2,\ 1)$
12. $(w,\ w_j,\ 1) \to (w,\ q_2,\ 1)$

\\Fault notifications on internal nodes (states q_2, w, and w_i), become l_1, which then introduce a new walking state.
13. $(l_1,\ q_2,\ 1) \to (e_1,\ w_1,\ 1)$

δ_2 :
$(e_i, 1) \to (q_0, 0)$
$(l_i, 1) \to (q_0, 0)$
$(q_2, 1) \to (l_1, 0)$
$(w, 1) \to (l_1, 0)$
$(w_i, 1) \to (l_1, 0)$

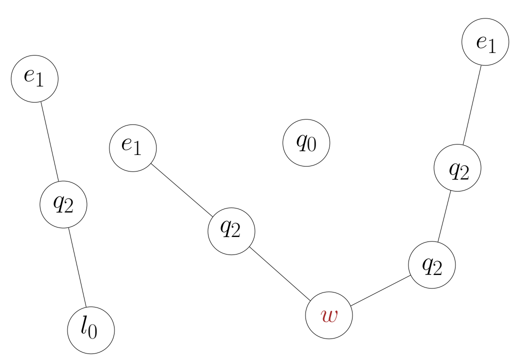


Figure 3: The left line is the result of one connection between two isolated nodes, and one expansion (rules 1 and 2 of Protocol 5). The second line is the result of a line merging (rule 3 of Protocol 5).

to e_2, w_2 and vice versa. In this way, we guarantee that w_i will eventually meet the other endpoint in state e_j, $j \neq i$, or l_0. In the first case, the w_i node becomes a leader (l_0), after having walked the whole line at least once. This process is described by rules $4-10$ of Protocol 5 (depicted also in Figure 4).

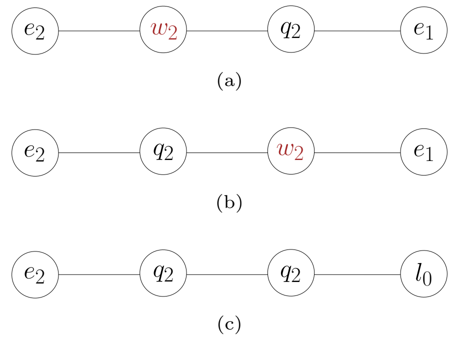


Figure 4: In Figure 4a, the walking state and the left endpoints are in states w_2 and e_2. Then, the walking state eventually reaches the second endpoint which is in state e_1, resulting to state l_0 (Figure 4c).

Now, consider the case where a fault may happen on some node on the line. If the fault flag of an endpoint state becomes 1, it updates its state to q_0. Otherwise, the line splits into two disjoint lines and the new endpoints become l_1. An l_1 becomes a walking state w_1, changes the endpoint into e_1 and performs a random walk (rule 13 of Protocol 5).

If there are more than one walking states on a line, then all of them are w, or w_i and they perform a random walk. None of them can ever satisfy the criterion to become l_0 before first eliminating all the other walking states and/or the unique leader l_0 (when two walking states meet, only one survives and becomes w), simply because they form natural obstacles between itself and the other endpoint (rules 11 and 12 of Protocol 5). This process is depicted in Figure 5. If a new fault occurs, then this can only introduce another w_i state which cannot interfere with what existing w_i's are doing on the rest of the line (can meet them eventually but cannot lead them into an incorrect decision).

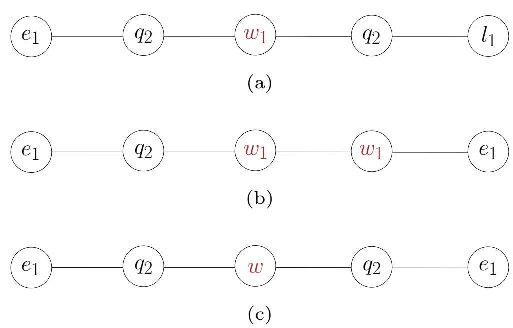


Figure 5: In Figure 5a there is one walking state, and one endpoint in state l_1. l_1 state is the result of a fault on an adjacent node of it. This state introduces an additional walking state, and in Figure 5c the two walking states interact and only one survives. The unique walking state w is then guaranteed to first traverse the whole line at least once before an endpoint becomes l_0.

If an l_0 leader is merging while there are w_i's and/or w's on its line (without being aware of that), the merging results in a new w state, which is safe because a w cannot make any further progress without first succeeding to beat everybody on the line. A w can become l_0 only after walking the whole line at least once (i.e., interact with both endpoints) and to do that it must have managed to eliminate all other walking states of the line on its way.

We have shown that despite the presence of faults, any expansion or merging eventually succeeds, meaning that the population eventually forms a line with a single leader in one endpoint. ☐

Lemma 7. *There is a NET* Π *(with notifications) such that when* Π *is executed on* n *nodes and at most* f *faults can occur, where* $0 \leq f < n$, Π *will eventually simulate a given TM* M *of space* $O(n-f)$ *in a fault-tolerant way.*

Proof. The state of Π has two components (P, S), where P is executing a spanning line formation procedure, while S handles the simulation of the TM M. Our goal is to eventually construct a spanning line, where initially the state of the second component of each node is in an initial state s_0 except from one node which is in state *head* and indicates the head of the TM.

In general, the states P and S are updated in parallel and independently from each other, apart from some cases where we may need to reset either P, S or both.

In order to form a spanning line under crash failures, the P component will be executing our *FT Spanning Line* protocol which is guaranteed to construct a line, spanning eventually the non-faulty nodes.

It is sufficient to show that the protocol can successfully reinitialize the state of all nodes on the line after a final *event* has happened and the line is stable and spanning. Such an *event* can be a line merging, a line expansion, a fault on an endpoint or an intermediate fault. The latter though can only be a final event if one of the two resulting lines is completely eliminated due to faults before merging again. In order to re-initialize the TM when the line expands to an isolated node q_0, we alter a rule of the *FT Spanning Line* protocol. Whenever, a leader l_0 expands to an isolated node q_0, the leader becomes q_2 while the node in q_0 becomes l_1, thus introducing a new walking state.

We now exploit the fact that in all these cases, *FT Spanning Line* will generate a w or a w_i state in each affected component.

Whenever a w_1 or w_2 state has just appeared or interacted with an endpoint e_1 or e_2 respectively, it starts resetting the simulation component S of every node that it encounters. If it ever manages to become a leader l_0, then it finally restarts the simulation on the S component by reintroducing to it the *tape head*.

When the last event occurs, the final spanning line has a w or w_i leader in it, and we can guarantee a successful restart due to the following invariant. Whenever a line has at least one w/w_i state and no further events can happen, *FT Spanning Line* guarantees that there is one w or w_i that will dominate every other w/w_i state on the line and become an l_0, while having traversed the line from endpoint to endpoint at least once.

In its final departure from one endpoint to the other, it will dominate all w and w_i states that it will encounter (if any) and reach the other endpoint. Therefore, no other w/w_i states can affect the simulation components that it has reset on its way, and upon reaching the other endpoint it will successfully introduce a new *head* of the TM while all simulation components are in an initial state s_0. □

Lemma 8. *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*.*

Proof. Initially all nodes are in state q_0. Protocol Π partitions the nodes into two equal sets U and D and every node maintains its type forever. This is done by a perfect matching between q_0's where one becomes q_u and the other becomes q_d. Then, the nodes of U execute the *FT Spanning Line* protocol, which guarantees the construction of a spanning line, capable of simulating a TM (Lemma 7). The rest of the nodes (D), which are connected to exactly one node of U each, are used to construct on them random graphs. Whenever a line merges with another line or expands towards an isolated node, the simulation component S in the states of the line nodes, as described in Lemma 7, is reinitialised sequentially.

Assume that a fault occurs on some node of the perfect matching before that pair has been attached to a line. In this case, its pair will become isolated therefore it is sufficient to switch that back to q_0.

If a fault occurs on a D node u after its pair z has been attached to a line, z goes into a detaching state which disconnects it from its line neighbors, turning them into l_1 and itself becoming a q_0 upon release. An l_1 state on one endpoint is guaranteed to walk the whole line at least once (as w_i) in order to ensure that a unique leader l_0 will be created. If u fails before completing this process, its neighbors on the line shall be notified becoming again l_1, and if one of its neighbors fails we shall treat this as part of the next type

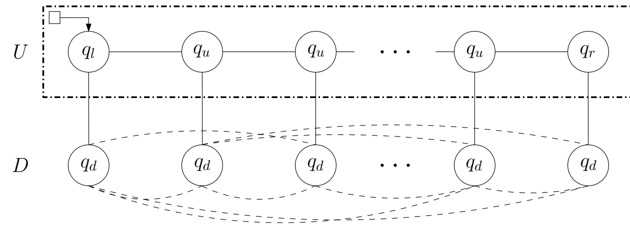


Figure 6: The population is partitioned into two groups U and D that form a perfect matching. The nodes of U eventually form a spanning line, and simulate a TM of linear space. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language.

of faults. This procedure shall disconnect the line but may leave the component connected through active connections within D. But this is fine as long as the *FT-Spanning Line* guarantees a correct restart of the simulation after any event on a line. This is because eventually the line in U will be spanning and the last event will cause a final restart of the simulation on that line.

Assume that a fault occurs on a node $u \in U$ that is part of the line. In this case the neighbors of u on the line shall instantly become l_1. Now, its D pair v, which may have an unbounded number of D neighbors at that point, becomes a special *deactivating state* that eventually deactivates all connections and never participates again in the protocol, thus, it stays forever as waste. This is because the fault partially destroys the data of the simulation, thus, we cannot safely assume that we can retrieve the degree of v and successfully deactivate all edges. As there can be at most $f(n)$ such faults we have an additional waste of $f(n)$. Now, consider the case where u is one neighbor of a node z which is trying to release itself after its v neighbor in D failed. Then, z implements a 2-counter in order to remember how many of its alive neighbours have been deactivated by itself or due to faults in order to know when it should become q_0. □

Theorem 7. *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.* [1]

Proof. By Lemma 8, there is a protocol that constructs two groups U and D of equal size, where each node of U is matched with exactly one node of D, and vice versa. In addition, the nodes of U form a spanning line, and by Lemma 7 it can simulate a TM M. After the last fault occurs, M is correctly initialized and the head of the TM is on one of the endpoints of the line. The two endpoints are in different states, and assume that the endpoint that the head ends up is in state q_l (*left* endpoint), and the other is in state q_r (*right* endpoint). This construction is depicted in Figure 6.

We now provide the protocol that performs the simulation of the TM M, which we separate into several subroutines. The first subroutine is responsible for simulating the direction on the tape and is executed once the head reaches the endpoint q_l. The simulation component S (as in Lemma 7) of each node has three sub-components (h, c, d). h is used to store the head of the TM, i.e., the actual state of the control of the TM, c is used to store the symbol written on each cell of the TM, and d is either l, r or $\sqcup$, indicating whether that node is on the left or on the right of the head (or unknown). Assume that after the initialization of the TM, $d = \sqcup$ for all nodes of the line. Finally, whenever the head of the TM needs to move from a node u to a node z, $h_z \leftarrow h_u$, and $h_u \leftarrow \sqcup$.

Direction. Once the head of the TM is introduced in the endpoint q_l by the lines' leader, it moves on the line, leaving l marks on the d component of each node. It moves on the nodes which are not marked, until it eventually reaches the q_r endpoint. At that point, it starts moving on the marked nodes, leaving r marks on its way back. Eventually, it reaches again the q_l endpoint. At that time, for each node on its right it holds

[1]Given a target graph of size $|V(G)|$, the size of the initial population required to construct G depends on the number of faults that occur and on the state of the nodes during the crash failures. In particular, the minimum size required to construct G is $2n$ (no faults occur), while the maximum number of nodes is $2(n + f(n))$.

that $d = r$. Now, every time it wants to move to the right it moves onto the neighbor that is marked by r while leaving an l mark on its previous position, and vice versa. Once the head completes this procedure, it is ready to begin working as a TM.

Construction of random graphs in D. This subroutine of the protocol constructs a random graph in the nodes of D. Here, the nodes are allowed to toss a fair coin during an interaction. This means that we allow transitions that with probability 1/2 give one outcome and with 1/2 another. To achieve the construction of a random graph, the TM implements a binary counter C ($\log n$ bits) in its memory and uses it in order to uniquely identify the nodes of set D according to their distance from q_l. Whenever it wants to modify the state of edge (i, j) of the network in D, the head assigns special marks to the nodes in D at distances i and j from the left of the endpoint q_l. Note that the TM uses its (distributed) binary counter in order to count these distances. If the TM wants to access the $i-$th node in D, it sets the counter C to i, places a mark on the left endpoint q_l and repeatedly moves the mark one position to the right, decreasing the counter by one in each step, until $C = 0$. Then, the mark has been moved exactly i positions to the right. In order to construct a random graph in D, it first assigns a mark r_1 to the first node q_l, which indicates that this node should perform random coin tosses in its next interactions with the other marked nodes, in order to decide whether to form connections with them, or not. Then, the leader moves to the next node on its line and waits to interact with the connected node in D. It assigns a mark r_2, and waits until this mark is deleted. The two nodes that have been marked (r_1 and r_2), will eventually interact with each other, and they will perform the (random) experiment. Finally the second node deletes its mark (r_2). The head then, moves to the next node and it performs the same procedure, until it reaches the other endpoint q_r. Finally, it moves back to the first node (marked as r_1), deletes the mark and moves one step right. This procedure is repeated until the node that should be marked as r_1 is the right endpoint q_r. It does not mark it and it moves back to q_l. The result is an equiprobable construction of a random graph. In particular, all possible graphs over $|D|$ nodes have the same probability to occur. Now, the input to the TM M is the random graph that has been drawn on D, which provides an encoding equivalent to an adjacency matrix. Once this procedure is completed, the protocol starts the simulation of the TM M. There are $m = k(k-1)/2$ edges, where $k = |D|$ and M has available $\frac{k}{2} = \sqrt{m}$ space, which is sufficient for the simulation on a $\sqrt{m}-$space TM.

Read edges of D. We now present a mechanism, which can be used by the TM in order to read the state of an edge joining two nodes in D. Note that a node in D can be uniquely identified by its distance from the endpoint q_l. Whenever the TM needs to read the edge joining the nodes i and j, it sets the counter C to i. Assume w.l.o.g. that $i < j$. It performs the same procedure as described in the subroutine which draws the random graph in D. It moves a special mark to the right, decreasing C by one in each step, until it becomes zero. Then, it assigns a mark r_3 on the $i-$th node of D, and then performs the same for $C = j$, where it also assigns a mark r_4 (to the $j-$th node). When the two marked nodes (r_3 and r_4) interact with each other, the node which is marked as r_4 copies the state of the edge joining them to a flag $\mathcal{F}$ (either 0 or 1), and they both delete their marks. The head waits until it interacts again with the second node, and if the mark has been deleted, it reads the value of the flag $\mathcal{F}$.

After a simulation, the TM either accepts or rejects. In the first case, the constructed graph belongs to L and the Turing Machine halts. Otherwise, the random graph does not belong to L, thus the protocol repeats the random experiment. It constructs again a random graph, and starts over the simulation on the new input.

A final point that we should make clear is that if during the simulation of the TM an event occurs (crash fault, line expansion, or line merging), by Lemma 7 and Lemma 8, the protocol reconstructs a valid partition between U and D, the TM is re-initialized correctly, and a unique head is introduced in one endpoint. At that time, edges in D may exist, but this fact does not interfere with the (new) simulation of the TM, as a new random experiment takes place for each pair of nodes in D prior to each simulation. $\square$

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 $k = n/3$. The idea is to use the

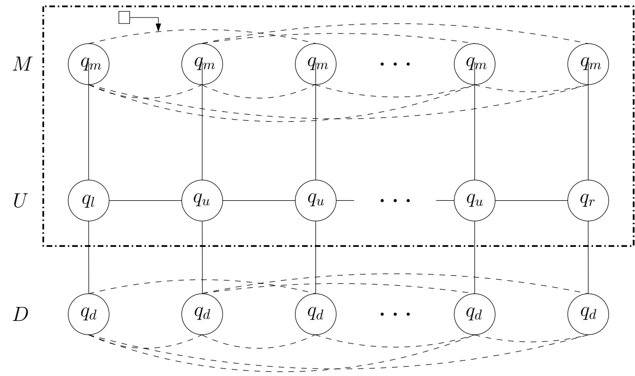


Figure 7: The population is partitioned into three groups M, U, and D of equal size. The nodes of U form a perfect matching with both U and M. The nodes of U eventually form a spanning line, and simulate a TM of linear space that uses the edges in M as an $O(n^2)$ binary memory. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language.

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.

Lemma 9. *Protocol 3-Partition partitions the nodes into three groups U, D and M, with waste $3f(n)$, 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, each node in $D \cup M$ is connected with exactly one node of U, and each node of U is connected to exactly one node in D and one node in M.*

Proof. Protocol 3−Partition constructs lines of three nodes each, where one endpoint is in state q_d, the other endpoint in state q_m, and the center is in state q_u. The nodes of U operate as in Lemma 8 (i.e., they execute the *FT Spanning Line* protocol). A (connected) pair of nodes waits until a third node is attached to it, and then the center becomes q_u and starts executing the *FT Spanning Line* protocol. Note that at some point, it is possible that the population may only consist of pairs in states q_d and q'_u. For this reason, we allow q'_u nodes to connect with each other, forming lines of four nodes. One of the q'_u nodes becomes q_u and the other becomes q'_m. A node in q'_m becomes q_m only after deactivating its connection with a q_d node (its previous pair). This results in lines of three nodes each with nodes in states q_d, q_u and q_m. Then, the q_u nodes start forming a line, spanning all nodes of U. In a failure-free setting, the correctness of this protocol follows from Lemma 8. In addition, by Lemma 7, the TM of the line is initialized correctly after the last occurring event (line expansion, line merging, or crash fault).

If we consider crash failures, it is sufficient to show that eventually U is a spanning line and M and D are disjoint. If a node ever becomes q_d or q_m, it might form connections with other nodes in D or M respectively, because of a TM simulation. A node in M never forms connections with nodes in D. After they receive a fault notification, they become the *deactivating state* s. A node in state s is disconnected from any other node, thus, it eventually becomes isolated and never participates in the execution again. We do this because nodes in M and D can form unbounded number of connections. The data of the TM have been partially destroyed (because of the crash failure), therefore it is not safe to assume that we can retrieve the degree of them and successfully re-initialize them.

A node u in state q'_m (inner node of a line of four nodes), after a fault notification it becomes q_w. A node in q_w waits until its next interaction with a connected node v. If v is in state q_u, this means that now a triple has been formed, thus u becomes q_m. If v is in state q_d, they delete the edge joining them, u becomes q_0 and v becomes s (v might have formed connections with other nodes in D).

A node u in q_u, after a fault notification it becomes q'_w and waits until its next interaction with a connected node v. At that point, v can be either q_d, q'_m, or q_m. In all cases they disconnect from each other and u becomes q'_0. The state q'_0 indicates that the node should release itself from the spanning line in U. This procedure works as described in Lemma 8, thus, after releasing itself from the line, it becomes q_0. If v is in state q_d or q_m, it becomes s. If v is in state q'_m, it becomes q'_u, as its (unique) adjacent node can only be in state q_d.

Protocol 6 3-Partition

$Q = \{q_0,\ q_0',\ q_d,\ q_u,\ q_u',\ q_m,\ q_m',\ q_w,\ q_w',\ s\} \times \{0, 1\}$
Initial state: q_0

δ_1 :
\\Formation of independent lines of three nodes.
$(q_0,\ q_0,\ 0) \to (q_u',\ q_d,\ 1)$
$(q_u',\ q_0,\ 0) \to (q_u,\ q_m,\ 1)$

\\Two connected pairs of nodes can form a line of four nodes. In this case, one of the endpoints is disconnected from the line.
$(q_u',\ q_u',\ 0) \to (q_u,\ q_m',\ 1)$
$(q_m',\ q_d,\ 1) \to (q_m,\ q_0,\ 0)$

\\q_w is the result of a fault on either a node in q_d or q_u state. The nodes in this state wait until they interact with (the unique) adjacent node, and update their state accordingly.
$(q_w,\ q_d,\ 1) \to (q_0,\ s,\ 0)$
$(q_w,\ q_u,\ 1) \to (q_m,\ q_u,\ 1)$

\\q_w' is the result of a fault on a node in q_d, q_m, or q_m' state. The nodes in this state wait until they interact with (the unique) adjacent node, and update their state accordingly. q_0' eventually becomes q_0 after releasing itself from the spanning line.
$(q_w',\ q_d,\ 1) \to (q_0',\ s,\ 0)$
$(q_w',\ q_m,\ 1) \to (q_0',\ s,\ 0)$
$(q_w',\ q_m',\ 1) \to (q_0',\ q_u',\ 0)$

\\Nodes in state s are disconnected from all nodes, and are left as waste.
$(s, \cdot,\ 1) \to (s, \cdot,\ 0)$

δ_2 :
$(q_u', 1) \to (q_0, 0)$

\\States q_m' and q_u indicate intermediate nodes of the line. After a fault notification, they enter to a temporary state, and wait until their first interaction with the remaining (unique) adjacent node.
$(q_m', 1) \to (q_w, 0)$
$(q_u, 1) \to (q_w', 0)$

\\A node in state q_w or q_w', is guaranteed to have exactly one neighbor in the (q_m, q_u, q_d) line. Thus, after a fault notification, it becomes q_0 (or q_0' if it belongs to the set U)
$(q_w, 1) \to (q_0, 0)$
$(q_w', 1) \to (q_0', 0)$

\\q_d and q_m nodes remain as waste (in state s) after a fault notification.
$(q_d, 1) \to (s, 0)$
$(q_m, 1) \to (s, 0)$

\\All transitions that do not appear have no effect.

A node in q_u' or q_w, after a fault notification it becomes q_0 and continues participating in the execution again. Finally, a node in state q_w', after receiving a fault notification, it becomes q_0' (a q_w' is the result of a fault notification in a $U-$ node).

Note that a node in any state except from q_d and q_m can be re-initialized correctly, thus they may participate in the execution again. It is apparent that no node that might have formed unbounded number of connections can participate in the execution again after a crash fault. This guarantees that the connections in D and M can be correctly initialized after the final event, and that no node in $D \cup M$ can be connected with more than one node in U. In addition, if a $U-$node receives a fault notification, it releases itself from the line, thus introducing new walking states in the resulting line(s). By Lemma 7, this guarantees the correct re-initialization of the TM. Finally, a crash failure can lead in deactivating two more nodes, in the worst case. These nodes never participate in the execution again, thus they remain forever as waste. This means that after $f(n)$ crash failures, the partitioning will be constructed in $n - 3f(n)$ nodes. □

Theorem 8. *For any graph language L that can be decided by an $(n^2/18 + O(n))-$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.*

Proof. Protocol 6 partitions the population in three groups U, D and M and by Lemma 9, it tolerates any number of crash failures, while initializing correctly the TM after the final event (line expansion, line merging, or crash fault). Reading and writing on the edges of M is performed in precisely the same way as reading/writing the edges of D (described in Theorem 7). Thus, the Turing Machine has now a $n^2/18-$space binary memory (the edges of M) and $O(n)-$space on the nodes of the spanning line U. The random graph is constructed on the k nodes of D (useful space), where by Lemma 9, $k = (n - 3f(n))/3 = n/3 - f(n)$ in the worst case. □

4.3. Designing Fault-Tolerant Protocols without Waste

A very simple, (yet impractical) idea that could tolerate any number $f < n$ of faults is to restart the protocol each time a node crashes. The implementation of this idea requires the ability of some nodes to *detect* the removal of a node.

Definition 8 (Global restart)**.** *Let Π be a protocol that constructs a graph language L and $\mathcal{C}$ be a set of configurations that all executions of Π starting from any $C \in \mathcal{C}$ stabilize to a graph $G \in L$. We call* global restart *the process which reaches Π to a configuration $C \in \mathcal{C}$ in finite time.*

Our goal is to come up with a protocol A that can be composed with any NET protocol Π (with notifications), so that their composition is a fault-tolerant version of Π. Essentially, whenever a fault occurs, A will restart all nodes in a way equivalent to as if a new execution of Π had started on the whole remaining population. Parallel execution of protocols is easily achieved in the Population Protocol model, by taking the Cartesian product of their state sets and updating the states for each protocol independently when a transition occurs ([3]). We denote the parallel composition of two protocols Π_1 and Π_2 as $\Pi_1 \circ \Pi_2$. However, in the Network Constructors model, the connections between the nodes are binary and the Cartesian product of their states would imply that each protocol Π_1 and Π_2 maintains its own connection state between each pair of nodes. To overcome this problem, we only consider parallel composition between two protocols where only one of them is allowed to activate/deactivate edges between the nodes. In particular, assume that Π_1 is a protocol with transition function $\delta_1 : Q_1 \times Q_1 \to Q_1 \times Q_1$ and Π_2 is a protocol with transition function $\delta_2 : Q_2 \times Q_2 \times \{0, 1\} \to Q_2 \times Q_2 \times \{0, 1\}$. Then, the composition of these protocols $\Pi_1 \circ \Pi_2$ is a Network Constructor with transition function $\delta_{1,2} : (Q_1 \times Q_2) \times (Q_1 \times Q_2) \times \{0, 1\} \to (Q_1 \times Q_2) \times (Q_1 \times Q_2) \times \{0, 1\}$.

Definition 9. *Consider any execution $\mathcal{E}_i$ of a protocol Π. There exists a finite number of different executions, and for each execution a step t_i that Π stabilizes. Call $C_{i,j}$ the $j-$th configuration of execution $\mathcal{E}_i$, where $j \leq t_i$. Then, we call maximum reachable degree of Π the value $d = max\{Degree(G(C_{i,j}))\}$, $\forall i, j$.*

We first show that even in the case where the whole population is notified about a crash failure, global restart is *impossible for protocols with $d = \omega(1)$, if the nodes have constant memory.* However, we provide a protocol that restarts the population, but we supply the nodes with $O(\log n)$ bits of memory. In our approach, we use fault notifications, and if a node z crashes, the set N_z of the nodes that are notified, has the task to restart the protocol.

Theorem 9. *Consider a protocol Π with unbounded maximum reachable degree. Then, global restart of Π is impossible for nodes with constant memory, even if every node u in the population is notified about the crash failure.*

Proof. Consider a protocol Π with constant number of states k and unbounded *maximum reachable degree*, which constructs a graph language L. Assume also that at time t a crash failure occurs and that there are some edges in the graph (call them *spurious edges*). Protocol Π is allowed to have rules that are triggered by the fault and try to erase those edges (*erasing process*). We assume that all nodes in the population are notified about the crash failure.

Observe that any degree more than k cannot be remembered by a node, that is, a state q cannot indicate its degree. This means that a node cannot detect the termination of the *erasing process* and eventually reset its state to an initial one to allow the restart. To stop the erasing process is equivalent to counting the remaining edges and wait until the degree reaches zero, but this would require logarithmic to *maximum reachable degree* number of bits.

The above observation means that the agents must enter to a new initial state and start forming new connections prior to the termination of the *erasing process.* But the only way to distinguish connections made before and after a fault is to enter to a different set of states after every a fault occurs. Otherwise the *erasing process* will fail. This can be achieved by having parallel executions of Π. In particular, given Π, the agents execute $\Pi' = \Pi^{(1)} \circ \Pi^{(2)} \circ \cdots \circ \Pi^{(\delta)}$, where $\Pi^{(i)}$ is obtained from Π by adding a constant i to it. Initially, the agents execute $\Pi^{(i)}$ for $i = 1$, and whenever a fault notification is received, the agents start executing $\Pi^{(i+1)}$.

As an example, consider the following protocol Π with initial state s and a single rule $(s, s) \to (r, r)$. Then $\Pi' = \Pi^{(1)} \circ \Pi^{(2)}$ has the following rules: $(s_1, s_1) \to (r_1, r_1)$ and $(s_2, s_2) \to (r_2, r_2)$, and the agents are initially in state s_1.

Assume that there exists an *erasing process* that can distinguish between edges made by different $\Pi^{(i)}$. Let $E^t_{u,i}$ be the set of activated edges of a node u at time t that were formed during the execution of $\Pi^{(i)}$. As the memory of each node is constant, then δ is also a constant number. In addition, the number of faults that can occur is unbounded, thus after δ faults at time t a node must execute a $\Pi^{(i)}$ that was executed in a previous step. However, $E^t_{u,i}$ might not be empty and then the *erasing process* will fail. □

In light of the impossibility result of Theorem 9, we allow the nodes to use non-constant local memory in order to develop a fault tolerating procedure based on restart.

We give a protocol that restarts any protocol Π as follows. All nodes are initially leaders. Through a standard pairwise leader elimination procedure, a unique leader would be guaranteed to remain in the absence of failures. But because a fault can remove the last remaining leader, the protocol handles this by generating a new leader upon getting a fault notification. This guarantees the existence of at least one leader in the population and eventually (after the last fault) of a unique one. There are two main events that trigger a new restarting phase: a fault and a leader elimination. As any new event must trigger a new restarting phase that will not interfere with an outdated one, eventually overriding the latter and restarting all nodes once more, we use phase counters to distinguish among phases. In the presence of a new event it is always guaranteed that a leader at maximum phase will eventually increase its phase, therefore a restart is guaranteed after any event. The restarts essentially cause gradual deactivation of edges (by having nodes remember their degree throughout) and restoration of nodes' states to q_0, thus executing Π on a fresh initial configuration. For the sake of clarity, we first present a simplified version of the restart protocol that guarantees resetting the state of every node to a uniform initial state q_0. So, for the time being we may assume that the protocol to be restarted through composition is any Population Protocol Π that always starts from the uniform q_0 initial configuration (all $u \in V$ in q_0 initially). Later on we shall extend this to handle with protocols that are Network Constructors instead.

Description of the PP Restarting Protocol. The state of every node consists of two components S_1 and S_2. S_1 runs the restarting protocol A while S_2 runs the given PP Π. In general, they run in parallel with the only exception when A restarts Π. The S_1 component of every node stores a *leader* variable, taking values from $\{l, f\}$, and is initially l, a *phase* variable, taking values from $\mathbb{N}_{\geq 0}$, initially 0, and a *fault* binary flag, initially 0.

The transition function is as follows. We denote by $x(u)$ the value of variable x of node u and $x'(u)$ the value of it after the transition under consideration.

If a leaders' fault flag becomes 1 or 2, it sets it to 0, increases its phase by one, and restarts Π. If a followers' flag becomes 1 or 2, it sets it to 0, increases its phase by one, becomes a leader, and restarts Π. We now distinguish three types of interactions.

When a leader u interacts with a leader v, one of them remains leader (state l) and the other becomes a follower (state f), both set their phase variable to $max\{\text{phase}(u), \text{phase}(v)\} + 1$ and both reset their S_2 component (protocol Π) to q_0 (i.e., restart Π).

When a leader u interacts with a follower v, if $\text{phase}(u) = \text{phase}(v)$, do nothing in S_1 but execute a transition of Π (both u and v involved). If $\text{phase}(u) < \text{phase}(v)$, then both set their phase variable to $max\{\text{phase}(u), \text{phase}(v)\} + 1$ and both restart Π, and finally, if $\text{phase}(u) > \text{phase}(v)$, then $\text{phase}'(v) = \text{phase}(u)$ and v restarts Π.

When a follower u interacts with a follower v, if $\text{phase}(u) = \text{phase}(v)$ do nothing in S_1 but execute transition of Π. If $\text{phase}(u) > \text{phase}(v)$, then v sets $\text{phase}'(v) = \text{phase}(u)$ and v restarts Π, and finally, if $\text{phase}(u) < \text{phase}(v)$, then u sets $\text{phase}'(u) = \text{phase}(v)$ and u restarts Π.

We now show that given any such PP Π, the above restart protocol A when composed as described with Π, gives a fault-tolerant version of Π (tolerating any number of crash faults).

Lemma 10 (Leader Election)**.** *In every execution of A, a configuration C with a unique leader is reached, such that no subsequent configuration violates this property.*

Proof. If after the last fault there is still at least one leader, then from that point on at least one more leader appears (due to the fault flags) and only pairwise eliminations can decrease the number of leaders. But pairwise elimination guarantees eventual stabilization to a unique leader. It remains to show that there must be at least one leader after the last fault. The leader state becomes absent from the population only when a unique leader crashes. This generates a notification, raising at least one follower's fault flag, thus introducing at least one leader. □

Call a *leader-event* any interaction that changes the number of leaders. Observe that after the last leader-event in an execution there is a stable unique leader u_l.

Lemma 11 (Final Restart)**.** *On or after the last leader-event, u_l will go to a phase such that phase$(u_l) >$ phase(u), $\forall u \in V' \setminus \{u_l\}$, where V' denotes the remaining nodes after the crash faults. As soon as this happens for the first time, let S denote the set of nodes that have restarted* Π *exactly once on or after that event. Then $\forall u \in V' \setminus S,\ v \in S$, an interaction between u and v results in $S \leftarrow S \cup \{u\}$. Thus, S will eventually be $S = V'$.*

Proof. We first show that on or after the last leader-event there will be a configuration in which $\text{phase}(u_l) > \text{phase}(u)$, $\forall u \in V' \setminus \{u_l\}$ and it is stable. As there is a unique leader u_l and follower-to-follower interactions do not increase the maximum phase within the followers population, u_l will eventually interact with a node that is in the maximum phase. At that point it will set its phase to that maximum plus one and we can agree that before that follower also sets its own phase during that interaction to the new max, it has been satisfied that $\text{phase}(u_l) > \text{phase}(u)$, $\forall u \in V' \setminus \{u_l\}$.

When the above is first satisfied, $S = \{u_l, u\}$ and $\text{phase}(u_l) = \text{phase}(u) > \text{phase}(v)$, $\forall v \in V' \setminus S$. Any interaction within S, only executes a normal transition of Π, as in S they are all in the same phase.

Any interaction between a $u \in V' \setminus S$ and a $v \in S$, results in $S \leftarrow S \cup \{u\}$, because interactions between followers in $V' \setminus S$ cannot increase the maximum phase within $V' \setminus S$, thus $\text{phase}(v) > \text{phase}(u)$ holds and the transition is: $\text{phase}'(u) = \text{phase}(v)$ and u restarts Π, thus enters S. It follows that S cannot decrease and any interaction between the two sets increases S, thus S eventually becomes equal to V'. □

Putting Lemma 10 and Lemma 11 together gives the aforementioned result.

Theorem 10. *For any such PP Π, it holds that $A \circ \Pi$ is a fault-tolerant version of Π.*

Lemma 12. *The required memory in each agent for executing protocol A is $O(\log n)$ bits.*

Proof. Initially all nodes are potential leaders, and they eliminate each other, moving to next phases at the same time. In the worst case, a single leader u will eliminate every other leader, turning them into followers, thus in a failure-free setting the phase of u becomes at most $n-1$. If we consider the case where crash faults may occur, each fault can result in notifying the whole population. This will happen if u was adjacent to every other node by the time it crashed. Thus, all nodes increase their phase by one and become leaders again. In the worst case, a single leader eliminates all the other leaders, thus, after the first fault, the maximum phase will be increased by $n-2$. The maximum phase than can be reached is $\sum_{i=0}^{k}(n-i) = O(kn)$, where k is the maximum number of faults that may occur ($k < n$). Thus, each node is required to have $O(\log n)$ bits of memory. □

NET Restarting Protocol (with Notifications). We are now extending the *PP Restarting Protocol* in order to handle any NET protocol Π (with notifications). Call this new protocol B. We store in the S_1 component of each node $u \in V$ a *degree* variable, that is, whenever a connection is formed or deleted, u increases or decreases the value of *degree* by one respectively. In addition, whenever the *fault flag* of a node u becomes one, it means that an adjacent node of it has crashed, thus it decreases *degree* by one. In the case of Network Constructors, the nodes cannot instantly restart the protocol Π by setting their state to the initial one q_0. By Theorem 9, it is evident that we first need to remove all the edges in order to have a successful restart and eventually stabilize to a correct network.

We now define an intermediate phase, called *Restarting Phase R*, where the nodes that need to be restarted enter by setting the value of a variable *restart* to 1 (stored in the S_1 component). As long as their degree is more than zero, they do not apply the rules of the protocol Π in their second component S_2, but instead they deactivate their edges one by one. Eventually their degree reaches zero, and then they set *restart* to 0 and continue executing protocol Π. We can say that a node u, which is in phase i ($\text{phase}(u) = i$), becomes available for interactions of Π (in S_2) only after a successful restart. This guarantees that a node u will not start executing the protocol Π again, unless its degree firstly reaches zero.

The additional Restarting Phase does not interfere with the execution of the *PP Restarting Protocol*, but it only adds a delay on the stabilization time.

Lemma 13. *The variable degree of a node u always stores its correct degree.*

Proof. In a failure-free setting, whenever a node u forms a new connection, it increases its *degree* variable by one, and whenever it deactivates a connection, it decreases it by one. In case of a fault, all the adjacent nodes are notified, as their *fault flag* becomes one. Thus, they decrease their *degree* by one. In case of a fault with no adjacent nodes, a random node is notified, and its *fault flag* becomes two. In that case, it leaves the value of *degree* the same. □

Theorem 11. *For any NET protocol Π (with notifications), it holds that $B \circ \Pi$ is a fault-tolerant version of Π.*

Proof. Consider the case where a node u (either leader or follower) needs to be restarted. It enters to the restarting phase in order to deactivate all of its enabled connections, and it will start executing Π only after its degree becomes zero (by Lemma 13 this will happen correctly), thus, Π always runs in nodes with no spurious edges (edges that are the result of previous executions). Whenever two connected nodes $u \in R$ and $v \notin R$, where R is the *Restarting Phase*, interact with each other, they both decrease their *degree* variable by one, and they delete the edge joining them. Obviously, this fact interferes with the execution of Π in node v (which is not in the restarting phase), but v is surely in a previous phase than u and will eventually also enter in R. This follows from the fact that a node in some phase i can never start forming new edges before it has successfully deleted all of its edges before. New edges are only formed with nodes in the same phase i.

The new *Restarting Phase* does not interfere with the states of the *PP Restarting Protocol*, thus the correctness of B follows by Lemma 10 and Lemma 11. □

Lemma 14. *The required memory in each agent for executing protocol B is $O(\log n)$ bits.*

Proof. The maximum value that the variable *degree* can reach is the *maximum reachable degree* (d) of protocol Π. Thus, by Lemma 12, the states that each node is required to have is $O(dkn)$. Both d and k are less that $n-1$, thus, $O(n^3)$ states $= O(\log n)$ bits. □

5. Conclusions and Further Research

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 all 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. Finally, in Section 4.3 we showed a protocol that restarts the population whenever a fault occurs, and to achieve it we empowered the agents with $O(\log n)$ memory. An immediate question here is whether we can achieve the same results by simulating the algorithm of [19] to handle crash failures. 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.

Acknowledgments We would like to thank David Doty and Mahsa Eftekhari for their valuable comments and suggestions during the development of this research work.

References

[1] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, March 2006.

[2] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, November 2007.

[3] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4):1–28, 2008.

[4] Joffroy Beauquier, Peva Blanchard, and Janna 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.

[5] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[6] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[7] Colin Cooper, Anissa Lamani, Giovanni Viglietta, Masafumi Yamashita, and Yukiko Yamauchi. Constructing self-stabilizing oscillators in population protocols. *Information and Computation*, 255:336–351, 2017.

[8] Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.

[9] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric 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.

[10] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim 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.

[11] Giuseppe Antonio Di Luna, Paola Flocchini, Taisuke Izumi, Tomoko Izumi, Nicola Santoro, and Giovanni Viglietta. Population protocols with faulty interactions: the impact of a leader. In *International Conference on Algorithms and Complexity (CIAC)*, pages 454–466. Springer, 2017.

[12] Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, pages 1–33, 2019.

[13] Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[14] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, Nov 1993.

[15] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, Jul 2001.

[16] Michael Fischer and Hong Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. *OPODIS*, vol 4305:395 – 409, 2006.

[17] Kyle Gilpin, Ara Knaian, and Daniela 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.

[18] Nabil Guellati and Hamamache 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.

[19] Rachid Guerraoui and Eric 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.

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

[21] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Mediated population protocols. *Theoretical Computer Science*, 412(22):2434–2450, May 2011.

[22] Othon Michail, George Skretas, and Paul G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

[23] Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.

[24] Othon Michail and Paul G. Spirakis. Network constructors: A model for programmable matter. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *SOFSEM 2017: Theory and Practice of Computer Science*, pages 15–34, Cham, 2017. Springer International Publishing.

[25] Othon Michail, Paul G. Spirakis, and Michail Theofilatos. Fault tolerant network constructors. In *Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 243–255. Springer, 2019.

[26] Ryu Mizoguchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6):451–460, 2012.

[27] David Peleg. As good as it gets: Competitive fault tolerance in network structures. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 35–46, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.