SUMMARY A self-stabilizing protocol is a protocol that achieves its intended behavior regardless of the initial configuration (i.e., global state). Thus, a self-stabilizing protocol is adaptive to any number and any type of topology changes of networks: after the last topology change occurs, the protocol starts to converge to its intended behavior. This advantage makes self-stabilizing protocols extremely attractive for designing highly dependable distributed systems on dynamic networks. While conventional self-stabilizing protocols require that the networks remain static during convergence to the intended behaviors, some recent works undertook the challenge of realizing self-stabilization in dynamic networks with frequent topology changes. This paper introduces some of the challenges as a new direction of research in self-stabilization.

1. Introduction

In the last decades, distributed computation attracted considerable attention and large-scale distributed systems were designed and developed. More recent distributed systems contain higher dynamics, which are caused, for example, by faults/recoveries or mobility of system components. This exalts the importance of adaptability of distributed systems to dynamic changes in networks: a distributed system is highly desired to continue to perform its task with adapting autonomously to dynamic changes in network environments. However, since dynamic changes in networks are unpredictable, autonomous adaptability is difficult to attain.

Self-stabilizing protocols (or algorithms) are considered as a promising class of protocols for attaining autonomous adaptability to dynamic changes in networks. A self-stabilizing protocol can achieve its intended behavior regardless of the initial configuration (i.e., global state) [5]. Thus, a self-stabilizing protocol is resilient to any number and any type of transient faults and is adaptive to any number and any type of topology changes of networks: after the last fault or topology change occurs, the protocol starts to converge to its intended behavior. These advantages make self-stabilizing protocols extremely attractive for designing highly dependable distributed systems. Self-stabilization has attracted a great deal of attention of researchers and practitioners working in the field of distributed systems. Especially, due to high adaptability to changes in networks, several works on self-stabilization are carried out for dynamic networks.

A self-stabilizing protocol can converge to its intended behavior from an arbitrary initial configuration, but the convergence is guaranteed only when the distributed system experiences no new fault or topology change during the convergence. When the system experiences a new fault or topology change before completing the convergence, the configuration immediately after the disturbance is regarded as an arbitrary one, which is regarded as a new initial configuration. Thus, in dynamic networks with frequent topology changes, self-stabilizing protocols may be quite inefficient and, what is worse, they might never converge to their intended behavior forever. Most of the works on self-stabilization for dynamic networks assume that the networks remain static during convergence to the intended behavior [1], [9]–[12], [15], [20], [23]. The self-stabilizing mutual exclusion protocol proposed in [2] also requires that networks remain static during its convergence. The distinct advantage of this protocol is that it can continue its intended behavior in spite of (loosely restricted) topology changes after the convergence.

Because of a rapid increase in the size of distributed systems and development of dynamic distributed systems such as P2P systems and mobile ad hoc systems, self-stabilizing protocols that can tolerate frequent faults/recoveries or topology changes are highly desired. Some recent works try to tackle the difficulty in applying self-stabilizing protocols to dynamic networks with frequent changes. In this paper, we introduce some challenges to attain self-stabilizing protocols for dynamic networks. The aim of this paper is not to give exhaustive survey but to introduce some challenges as a new direction of research on self-stabilization.

Organization of This Paper

After some preliminary definitions in Sect. 2, three approaches to stabilization in dynamic networks are presented using examples of the mutual exclusion in Sect. 3. The first one is dynamic reconfiguration tolerance proposed by Kakugawa and Yamashita [14] as a formulation of self-stabilizing protocols in dynamic networks. In the formulation, restriction on topology changes is explicitly specified. The second one is the mutual exclusion protocol proposed by Chen and Welch [3], which relaxes the assumption of their previous work [2] so that convergence can be attained despite dynamic changes. Instead of explicit re-
striction on topology changes, topology changes are implicitly restricted as performance requirement for an underlying protocol: the mutual exclusion protocol works well if an underlying token circulation protocol satisfies some criteria on performance. Similar implicit restriction is used in the self-stabilizing group communication protocol of [8]. The last one is a trial as a general approach to prove convergence in dynamic networks [19]. The approach tries to apply a commonly used technique, a potential function, to dynamic networks.

Section 4 introduces two concepts, superstabilization and safe convergence, as those important to dynamic networks. These concepts do not consider topology change during convergence to the intended behavior, but try to realize features suitable for dynamic networks. Finally, Sect. 5 contains some concluding remarks.

2. Model

2.1 Dynamic Networks

A network is denoted by \( N = (P, L) \) where \( P \) and \( L \) are respectively the set of processes (or nodes) and the set of (communication) links. A link in \( L \) connects two distinct processes in \( P \), and processes are called neighbors when they are connected by a link. Each process has a unique identifier and can identify its neighbors. Neighbors can directly communicate with each other. As communication models, we consider two models in this paper: a message passing model where each process can directly exchange messages with its neighbors, and a shared state model where each process can directly read the states (or variables) of its neighbors.

In this paper, we consider dynamic networks where its process set and its link set vary with time. We assume that each process can identify its neighbors at any time. Examples of dynamic networks contain P2P networks and mobile ad hoc networks, where processes can join and leave at any time and connection between processes change according to the locations of processes. Dynamic changes in topology is also caused by faults and recoveries of processes and links. A stopping fault of a process (resp. a link) can be considered as removal of the process and its incident links (resp. the link). A recovery of a process (resp. a link) from faults can be treated as addition of the process and its incident links (resp. the link).

2.2 Self-Stabilization

For dynamic networks, protocols highly adaptive to dynamic changes in topology are greatly desired. Self-stabilizing protocols (or algorithms) are promising candidates of adaptive protocols for dynamic networks. A self-stabilizing protocol is a protocol that achieves its intended behavior regardless of the initial configuration (or global state). More precisely, a self-stabilizing protocol is defined as follows [5].

**Definition 1:** A protocol is a self-stabilizing protocol for a task if any execution starting from an arbitrary configuration eventually reaches a safe configuration. A configuration \( \sigma \) is safe for a task and a protocol if every execution of the protocol starting from \( \sigma \) is a legal execution of the task.

Here, a task is defined by a set of “desired” executions called legal executions. As an example, we show the definition of legal executions of the mutual exclusion task for dynamic networks: An infinite execution \( E \) is legal for the mutual exclusion task if \( E \) satisfies the following two conditions:

- **Safety:** At every configuration in \( E \), at most one process has a privilege (to enter a critical section).
- **Liveness:** Every process has a privilege infinitely often in \( E \) if \( p \) eventually remains in the network.

The mutual exclusion is one of the most fundamental, and thus, the most investigated tasks in distributed systems. This paper also treats the mutual exclusion as an example in the next section.

We show, as an example, a self-stabilizing protocol for the mutual exclusion. Figure 1 shows a mutual exclusion protocol on static rings proposed by Dijkstra [4]. The ring is composed of processes \( p_0, p_1, \ldots, p_{n-1} \) and \( p_i \) \((0 \leq i \leq n-1)\) can directly read variables of \( p_{(i-1) \mod n} \) (shared state model). Each process \( p_i \) has a variable \( s_i \) that takes a non-negative integer in \([0, 1, \ldots, K-1]\) where \( K \) is an arbitrary constant no smaller than \( n \). Process \( p_0 \) is a special process called the token initiator and behaves differently from other processes. The actions of the token initiator \( p_0 \) and of other processes are described in Fig. 1 (a) and Fig. 1 (b) respectively.

The protocol works in asynchronous systems where no assumption is made for the timing each process makes actions.

Figure 2 shows an execution of Dijkstra’s protocol. The execution can be regarded as token circulation where a token represents a privilege. Process \( p_0 \) has a token of label \( s_0 \) if and only if \( s_0 = s_{p_0-1} \) holds (Fig. 1 (a)). Then, \( p_0 \) increments the label of the token and forwards the token to \( p_1 \) by executing \( s_0 := s_0 + 1 \). Each process \( p_i \) \((i \neq 0)\) has a token of label \( s_{p_i-1} \) if and only if \( s_i \neq s_{p_i-1} \) and it forwards the token of the same label to \( p_{(i+1) \mod n} \) by executing \( s_i := s_{p_i-1} \) (Fig. 1 (b)). If \( s_{p_{i+1}} = s_{p_i} \) \((i+1 \neq n)\) happens to
hold when \( p_i \) executes \( st_i := st_{i-1}, p_{i+1} \) does not hold the token in the resulting configuration. In this case, we consider the token forwarded by \( p_i \) is discarded. Similarly, if \( p_0 \) does not hold the token in the configuration resulting from execution of \( st_{n-1} := st_{n-2} \), we consider the token forwarded by \( p_{n-1} \) is discarded. Only \( p_0 \) can initiate a token with a new label incremented by one (Fig. 1 (a)), and every other processes \( p_i \) \((i \neq 0)\) only relays tokens by copying the value of \( st_{i-1} \) to \( st_i \) (Fig. 1 (b)). This is the reason we call \( p_0 \) the token initiator.

A configuration of Dijkstra’s protocol is safe if there exists \( i \) \((0 \leq i \leq n - 1)\) that satisfies the followings:

- For any \( j \) \((0 \leq j \leq i)\), \( st_j = st_0 \), and
- for any \( j \) \((i < j \leq n - 1)\), \( st_j = (st_0 - 1) \mod K \).

In the above safe configuration, only \( p_{i+1} \mod n \) has a token. In the execution of Fig. 2, configurations \( \alpha_0 \), \( \alpha_1 \), \( \alpha_2 \) and \( \alpha_3 \) are safe ones.

### 3. Stabilization Despite Dynamic Changes

#### 3.1 Dynamic Reconfiguration Tolerance

When dynamic networks experience too frequent changes in topology, no self-stabilizing protocol can converge to safe configurations. Thus, it is necessary to make some restrictions on topology change of dynamic networks, and several works made restrictions in different ways.

Kakugawa and Yamashita [14] introduced dynamic reconfiguration tolerance, as a generalization of conventional self-stabilization, to formalize self-stabilization under restricted dynamic changes in topology. Restriction on topology changes is explicitly specified by dynamic network reconfiguration constraint \( C \), and a self-stabilizing protocol is said to be dynamic reconfiguration tolerant under \( C \) if it satisfies the requirements of self-stabilizing protocols as long as network topology changes follow \( C \). The constraint \( C \) restricts dynamic changes of a network, e.g., “for some constant \( k \), any two changes in network topology should be separately in time at least \( k \) time units.” More precisely, dynamic reconfiguration tolerance is defined as follows.

**Definition 2:** A self-stabilizing protocol is said to be dynamic reconfiguration tolerant (DRT for short) under a dynamic network reconfiguration constraint \( C \) if the following two conditions are satisfied.

- **Convergence:** Any execution starting from an arbitrary configuration eventually reaches a safe configuration, as long as network topology changes follow \( C \).
- **Safety:** Any execution starting from any safe configuration is a legal execution, as long as network topology changes follow \( C \).

If we take a constraint “no network topology change occurs” for \( C \), a DRT self-stabilizing protocol under \( C \) is a conventional self-stabilizing one. Thus, DRT self-stabilization is a generalization of conventional self-stabilization.

We briefly describe the DRT self-stabilizing mutual exclusion protocol [14]. They adopt the message passing model as the communication model; each process can communicate with its neighbors by exchanging messages. The model is synchronous in the sense that messages experience bounded delay to transit a link and the processing time of local computation can be ignored. The upper bound of transmission delay is known to processes. Notice that the assumption of bounded delay is necessary from the following reason: Self-stabilizing protocols accept arbitrary initial configurations, that is, any number of illegal messages can be in transit in an initial configuration. The assumption of bounded delay keeps us from arbitrarily late influence of the illegal messages and from unbounded waits for non-existing messages.

The protocol proposed in [14] is a DRT self-stabilizing mutual exclusion protocol under the constraint that “no two link removals occur within \( 6(n - 1) \) time units and the network is not partitioned,” where \( n \) is the number of processes. The protocol is based on token circulation and can be regarded as application of Dijkstras’s protocol to a spanning tree of the network.

They assume that each link has a weight that is defined based on, for example, bandwidth, transmission delay, or reliability of the link. The protocol constructs and updates a spanning tree, and applies Dijkstra’s protocol to the spanning tree. The spanning tree constructed first is a depth-first-search tree and is updated so that the total weight of tree links are gradually decreased. Since an important and
attractive feature of the protocol is in construction and management of the spanning tree, we present more details of this part below. On the other hand, application of Dijkstra’s protocol to the spanning tree is carried out in a relatively simple way, so we omit the details of this part.

We assume for simplicity that a single process, say \( p_0 \), acts as a special process, a token initiator (as in Dijkstra’s protocol), while the original protocol chooses the process with the minimum identifier as the token initiator. The token initiator starts circulation of a token along a spanning tree in a depth-first fashion. The spanning tree is also constructed and maintained by the protocol itself. To construct a spanning tree, the token carries a tree (in terms of the tree link set) that spans the processes it has visited; it carries an empty link set when initiated, and a link \((p_i, p_j)\) is added to the set when a process \( p_i \) receives the token first from a process \( p_j \). Then, \( p_i \) forwards the token to an unvisited neighbor if exists. If all the neighbors of \( p_i \) have been already visited, \( p_i \) sends the token back to \( p_j \). The spanning tree constructed as above is a depth-first search tree and then it is gradually improved to form the minimum weight spanning tree.

To improve the current spanning tree, the token also carries information of a link that can improve the weight of the spanning tree. In case that two or more links can improve the weight of the spanning tree, the link with the minimum weight is carried by the token. The token initiator updates the spanning tree at each time the token completes the circulation. Thus, the weight of the spanning tree is gradually improved, and the minimum weight spanning tree is eventually obtained.

In static networks, the above strategy works well, however in dynamic networks the token circulation may not complete because of removal of tree links. The protocol circumvents the difficulty in the same way as Dijkstra’s protocol. The token initiator repeatedly generates tokens of the same label in some time intervals until a token of the label completes circulation over the whole network. To avoid the configuration where multiple tokens of the same label exist, each token has lifetime and the token is discarded when the lifetime expires. When the token initiator receives a token that completes circulation of the whole network, it starts initiating a token with a new label incremented by one (with some modulo).

Kakugawa and Yamashita [14] showed that the protocol, with token lifetime \( 6(n - 1) \) and token recreation interval \( 12(n - 1) \), is a DRT self-stabilizing mutual exclusion protocol under a dynamic network reconfiguration constraint that no two link removals occur within \( 6(n - 1) \) time units and the network is not partitioned. They also show that the protocol can attain a weaker liveness property under a weaker constraint on topology changes: under the constraint that no two link removals occur within \( 4(n - 1) \) time units and the network is not partitioned, the protocol, with token lifetime \( 6(n - 1) \) and token creation interval \( 12(n - 1) \), is a DRT self-stabilizing mutual exclusion protocol in a weak sense that an exactly one token is circulated but it may not visit all processes in each traversal.

3.2 Implicit Restriction on Topology Changes

Chen and Welch [3] proposed a self-stabilizing mutual exclusion protocol for mobile ad hoc networks and showed that the protocol attains different levels of progress depending on performance of an underlying token circulation protocol. Performance of the underlying protocol is affected by topology changes, thus, topology changes are implicitly restricted as the performance requirements.

The ad hoc networks are modeled as message passing models with bounded delay of message transmission. The protocol is also based on token circulation and can be regarded as application of Dijkstra’s protocol [5] to a virtual ring. The virtual ring is constructed and maintained by a self-stabilizing version ss-LRV of a token circulation protocol LRV [18]. Protocol ss-LRV circulates a token in the least-recently-visited fashion; a token is forwarded to the neighbor that was visited least recently by the token. To determine the least-recently-visited neighbor, a token carries a timestamp array where an entity is prepared for each process to stores the timestamp when the process is visited by the token.

Simulation of protocol LRV shows that the protocol performs well in mobile ad hoc networks. In particular, the number of hops for a token to complete a circulation of all the processes is very close to the optimal value \( n \), where \( n \) is the number of processes in the network. It is also proved that LRV requires exponential time in the worst case for completing a token circulation on some class of graphs.

The self-stabilizing mutual exclusion protocol assumes that a single token initiator, say \( p_0 \), exists. The token initiator \( p_0 \) initiates token circulation using ss-LRV. Since tokens may be lost due to leaves or mobility of processes, \( p_0 \) generates tokens of the same label repeatedly. The protocol allows multiple tokens of the same label to coexist. To guarantee the safety property of the mutual exclusion, all the tokens of the same label carry the same virtual ring and processes can get privileges at most once for each label and in the order of the virtual ring.

When a token completes circulation of the network, \( p_0 \) generates a token of a new label (incremented by one). In creation of the token, \( p_0 \) constructs a virtual ring and attaches it to the token. The virtual ring is constructed using the trace of a previous circulation each token carries with it.

The self-stabilizing mutual exclusion protocol eventually guarantees the safety property (i.e., at most one process has a privilege at the same time) under arbitrary topology changes, and guarantees that different levels of progress depending on performance of ss-LRV can achieve. The following three levels of progress are considered.

1. No deadlock mutual exclusion requires that some process has privileges infinitely often in any infinite exe-
2. **Dynamic no lockout** mutual exclusion requires that every process has privileges infinitely often in any infinite execution if it eventually remains in the network.

3. **Dynamic bounded waiting** mutual exclusion requires that there exists time interval $T$ such that every process has a privilege exactly once in every time interval $T$ if it remains in the network during the period.

Process $p$ has a privilege only when $p$ has a token. Thus the level of progress provided by the protocol depends on the frequency each process receives tokens, that is, $ss$-LRV’s performance for token circulation. They investigated requirements on $ss$-LRV’s performance to achieve the different levels of progress. Roughly speaking, the requirements they considered are as follows.

1. **No deadlock** is guaranteed if $ss$-LRV satisfies the following requirement: if infinitely many tokens are generated by $p_0$, then infinitely many of them visit some process and come back to $p_0$.

2. **No lockout** is guaranteed if $ss$-LRV satisfies the following requirement: If infinitely many tokens are generated by $p_0$, every process receives infinitely many of them and the tokens come back to $p_0$.

3. **Dynamic bounded waiting** is guaranteed if $ss$-LRV satisfies the following requirement: At least one of tokens generated by $p_0$ in every time interval $T$ completes circulation on the virtual ring.

The protocol in [3] has a drawback in fairness: since processes get a privilege in the order of the virtual ring, processes in earlier part of the ring can get a privilege more easily than those in latter part. Nishikawa et al. [22] pointed out this drawback of the protocol and modifies construction of the virtual ring so that fairness should be improved without sacrificing the overall performance.

### 3.3 Analysis of Stabilization on Dynamic Networks

It is generally said that self-stabilizing protocols are inefficient in dynamic networks with frequent topology changes. Its main reason is that a new topology change brings the system into an unexpected configuration, and thus, the system restarts convergence to its intended behavior from scratch. But the reasoning seems too pessimistic; Should a single topology change spoil all the efforts a self-stabilizing protocol made before the disturbance? This is the motivation of investigating analysis of stabilization on dynamic networks [19], [21].

The insight into the question is as follows. Each action of a self-stabilizing protocol can be regarded as a forward step to convergence, but a topology change can be regarded as a backward step. If we can quantify influence of the forward step and the backward step, we can estimate efficiency of the convergence with considering topology changes during the convergence. In other words, by evaluating the degree of regression a topology change can bring about, we can evaluate the total number of the protocol’s steps required to complete the convergence despite the disturbance.

The above strategy is applied to a simple extension of Dijkstra’s protocol for dynamic rings [19]. In the dynamic rings, processes can join and leave the ring at any time (Fig. 3). In case of a join, the joining process, say $p$, is inserted between two neighboring processes, say $q$ and $r$. In case of a leave, the neighbors, say $q$ and $r$, of the leaving process become neighbors of each other to keep forming a ring.

Each process $p$ in the ring can identify its predecessor $pred(p)$ in the ring at any instance. We assume that the adequate update of $pred(p)$ is atomically executed when a process joins or leaves 1. We assume the asynchronous shared state model and $p$ can read variables of $pred(p)$. We also assume that there exists a single designated process, the token initiator, at any instance. The token initiator remains the token initiator until it leaves the ring. When the token initiator $p$ leaves the ring, its successor $q$ (the process $q$ such that $pred(q) = p$) becomes the new token initiator.

The protocol is a simple extension of Dijkstra’s protocol; only actions executed for a process join and a process leave are added. The extension part is as follows:

1. **Join:** When a process $p$ newly joins the ring, it copies the value of variable $st$ from its new predecessor; it executes $st_p := st_{pred(p)}$. Then, $p$ starts to behave as an ordinary process (not as the token initiator).

2. **Leave:** When the token initiator, say $p$, leaves the ring, its successor, say $q$, copies the value of variable $st$ from $p$; it executes $st_q := st_p$. Then, $q$ starts to behave as the token initiator.

When a process other than the token initiator leaves, there is no extended code to execute. The neighbors of the leaving process are directly connected to form a ring, and simply continue to execute the code of Dijkstra’s protocol.

The safe configurations of Dijkstra’s protocol (described in Sect. 2) are also safe for dynamic rings. To show this, we need some restriction on frequency of joins from the following reason. At a configuration where process $p$ is privileged, the privilege moves to $p$’s successor when $p$ executes its action. By consecutive executions of actions of privileged processes, the privilege circulates along the ring.

---

1 The details of updating $pred(p)$ is not considered in [19], but maintenance of a dynamic ring is investigated in some works [16], [17].
ring. However, on a dynamic ring, the privilege cannot complete the circulation if joins have higher frequency than the actions of processes. Thus, the safe configurations of Dijkstra’s protocol are also safe for dynamic rings under the following restriction on frequency of joins: for any configuration $\sigma$ in an execution $E$, there exists a configuration $\sigma'$ appearing after $\sigma$ in $E$ such that the number of process actions executed between $\sigma$ and $\sigma'$ is larger than that of joins executed between $\sigma$ and $\sigma'$.

To show convergence of the protocol to a safe configuration, we use a potential function on configurations. The potential function defines a non-negative value on every configuration such that
- its value decreases by at least one when a process executes an action and
- when its value is zero for a configuration $\sigma$, then $\sigma$ is a safe configuration.

Potential functions are commonly used to prove convergence to a safe configuration and estimate the time required for the convergence in static networks. We developed and applied the potential function for dynamic rings, and showed that any leave action cannot increase the function value and that any join action can increase the function value by at most $2n$, where $n$ is the number of processes in the initial configuration. From the above observation, we can show that any execution reaches a safe configuration if there exists an execution fragment such that the number of process actions executed is $1.5n^2 + 2m$ or more. Here, $n$ denotes the number of processes in the initial configuration of the fragment and $m$ denotes the number of joins in the fragment.

4. Superstabilization and Safe Convergence

This section briefly introduces superstabilization and safe convergence. These concepts do not consider topology change during convergence to the intended behavior, but try to realize features suitable for dynamic networks.

4.1 Superstabilizing Protocols

Superstabilizing protocols are introduce by Dolev and Herman [6], [7] as self-stabilizing ones with an additional advantage: it can converge to a safe configuration in a graceful manner when started from a safe configuration and a topology change occurs. During convergence after the topology change, the protocols should preserve a passage predicate. When safety condition may depend on network topology, a topology change may falsify the safety condition. Thus, the passage predicate must be weaker than the safety, but strong enough to be useful. Notice that superstabilization is specified with a class of topology changes for which a desired passage predicate should be preserved during convergence, but guarantees nothing about convergence from any other topology change.

Dolev and Herman [6], [7] presented superstabilizing protocols for process coloring and spanning tree construction, and proposed a transform technique of self-stabilizing protocols to superstabilizing ones. Figure 4 presents a superstabilizing protocol for process coloring [7]. The protocol assigns a color to each process so that any neighboring processes should be assigned different colors. Each process $p$ has a variable $\text{color}_p$ to store its assigned color and a variable $\text{UsedCol}_p$ to store the assigned colors of neighbors. The color is taken from a color set $\{0, 1, \ldots, \Delta\}$ where $\Delta$ is an upper bound of the process degree (or the number of neighbors of a process). Variable $\text{color}_p$ can take a special value $\bot$ to specify $p$ is not assigned any color.

The protocol utilizes processes’ identifiers as priorities; when a process $p$ has the same color as its neighbor with a larger identifier (or higher priority), then $p$ resolves the color conflict by changing its color to a color not used by its neighbors (Fig. 4(a) line 8). To detect the color conflict with a process of higher priority, each process has a variable $\text{HUsedCol}_p$ that stores the colors assigned to its neighbors with higher priority. When link $(p, q)$ is added, $p$ resets its color by executing $\text{color}_p := \bot$ if $p$ has a smaller identifier (or lower priority) than $q$ (Fig. 4(b)). This action of Fig. 4(b) is the key action that enables the protocol to preserve the passage predicate.

It is easy to see that the protocol of Fig. 4 is a self-stabilizing protocol for process coloring. The passage predicate for superstabilization is that any neighbors never have the same color at any configuration. The protocol satisfies the passage predicate even during convergence to a safe configuration from a configuration resulting from a safe one by adding a single link or removing processes and links.

4.2 Self-Stabilization with Safe Convergence

Kakugawa and Masuzawa [13] introduced the concept of safe convergence of self-stabilizing protocols. The concept requires that the safety of a problem is guaranteed even during convergence. Such a concept implicitly appears in several previous papers, and is formulated in [13].

The safe convergence is defined with two classes of safe configurations. One is a set of configurations, called
feasible safe configurations, in which property or service is feasible, i.e., minimum quality of service is guaranteed. The other is a set of configurations, called optimal safe configurations, in which property or service is optimal. The safe convergence requires that the protocol should converge to a feasible safe configuration as soon as possible from an arbitrary configuration. If the network remains static for sufficiently long period, the protocol should converge to an optimal safe configuration to provide the best quality of service. The safety convergence property requires that the system should not break a safety property while moving from a feasible safe configuration to an optimal safe configuration.

The concept of the safe convergence seems similar to superstabilization. The considerable difference is that the safe convergence tries to guarantee a weak safety property during convergence from an arbitrary configuration while the superstabilization does that only during convergence from specific topology changes.

A self-stabilizing protocol with the safe convergence for the minimal independent dominating set (MIDS) problem is presented in [13]. A process set is a dominating set if each process is included in the set or has a neighbor in the set (a dominating set). A dominating set is said to be minimal if none of its proper subset is a dominating set. If no processes in a minimal dominating set are neighbors (an independent set), then it is called a minimal independent dominating set.

The (minimal) dominating set problem is a fundamental problem in ad hoc networks and has several applications. The most common application is clustering; each dominator (or process in the dominating set) becomes a cluster head and each dominante (or process not in the dominating set) belongs to a cluster whose head is a neighbor. Such clusters can be utilized for efficient communication in ad hoc networks, and should be maintained in response to mobility of mobile terminals.

In these applications, quick construction of a dominating set is essential for availability of communication. From a view point of efficiency, a smaller number of dominators may realize better efficiency, so the number of dominators should be reduced while keeping the availability of communication. Thus, a configuration with a minimal (independent) dominating set should be regarded as an optimal safe configuration, and a configuration with a dominating set should be regarded as a feasible safe configuration.

The protocol of [13] is a self-stabilizing MIDS protocol with safe convergence for the feasible safe configuration: any execution from an arbitrary configuration reaches a feasible safe configuration in one synchronous step and then eventually reaches an optimal safe configuration with keeping the condition of the feasible safe configuration.

5. Conclusions

Self-stabilizing protocols inherently have high adaptability to topology changes in dynamic networks. However, conventional self-stabilizing protocols require that networks remain static during convergence to a safe configuration. This paper introduced some recent works that try to tackle the difficulty in realizing self-stabilization on frequently changing networks.

Other than the works presented in this paper, several researches are carried out for self-stabilization in dynamic networks. Among them, especially, randomized protocols are expected to play an important role in this field. However, the theoretical fundamentals are not established yet even for deterministic approaches, and more investigation is required.

References

[1] H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. El-Ghazawi, “A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks,” J. Parallel Distrib. Comput., vol.63, pp.97–104, 2003.
[2] Y. Chen and J. Welch, “Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks,” Proc. 6th Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M), pp.34–42, 2002.
[3] Y. Chen and J. Welch, “Self-stabilizing dynamic mutual exclusion for mobile ad hoc networks,” J. Parallel Distrib. Comput., vol.65, pp.1072–1089, 2005.
[4] E. Dijkstra, “Self-stabilizing systems in spite of distributed control,” Commun. ACM, vol.17, no.11, pp.643–644, 1974.
[5] S. Dolev, Self-stabilization, The MIT Press, 2000.
[6] S. Dolev and T. Herman, “Superstabilizing protocols for dynamic distributed systems,” Proc. 2nd Workshop on Self-stabilizing Systems, pp.3.1–3.15, 1995.
[7] S. Dolev and T. Herman, “Superstabilizing protocols for dynamic distributed systems,” Chicago Journal of Theoretical Computer Science, 1997.
[8] S. Dolev, E. Schiller, and J. Welch, “Random walk for self-stabilizing group communication in ad-hoc networks,” Proc. 21st Symposium on Reliable Distributed Systems (SRDS), pp.70–79, 2000.
[9] W. Goddard, S. Hedetniemi, D. Jacobs, and P. Srimani, “Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks,” Proc. International Parallel and Distributed Processing, p.162b, 2003.
[10] S. Gupta, A. Bouabdallah, and P. Srimani, “Self-stabilizing protocol for shortest path tree for multicast routing in mobile networks,” Proc. 6th Euro-Par, pp.600–604, 2000.
[11] S. Gupta and P. Srimani, “Self-stabilizing multicast protocols for ad hoc networks,” J. Parallel Distrib. Comput., vol.63, pp.87–96, 2003.
[12] C. Johnen and L. Nguyen, “Self-stabilizing clustering algorithm for ad hoc networks,” Proc. 2nd International Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors), pp.83–93, 2006.
[13] H. Kakugawa and T. Masuzawa, “A self-stabilizing minimal dominating set algorithm with safe convergence,” Proc. 8th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM), Paper number 103, 2006.
[14] H. Kakugawa and M. Yamashita, “A dynamic reconfiguration tolerant self-stabilizing token circulation algorithm in ad-hoc networks,” Proc. 8th International Conference on Principles of Distributed Systems (OPODIS), pp.179–186, 2004.
[15] Y. Katayama, T. Hasegawa, and N. Takahashi, “A super-stabilizing spanning tree protocol for a link failure,” IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J88-D-I, no.11, pp.1669–1678, Nov. 2005.
[16] X. Li, J. Misra, and C. Plaxton, “Active and concurrent topology maintenance,” Proc. 18th International Symposium on Distributed Computing (DISC), pp.320–334, 2004.
[17] X. Li, J. Misra, and C. Plaxton, “Brief announcement: Concurrent maintenance of rings,” Proc. 23rd Symposium on Principles of Dis-
[18] N. Malpani, Y. Chen, N. Vaidya, and J. Welch, “Distributed token circulation in mobile ad hoc networks,” IEEE Trans. Mobile Computing, vol.4, no.2, pp.154–165, 2005.

[19] T. Masuzawa and H. Kakugawa, “Self-stabilization in spite of frequent changes of networks: Case study of mutual exclusion on dynamic rings,” Proc. 7th International Symposium on Self-Stabilizing Systems (SSS), pp.183–197, 2005.

[20] S. Miyanaga, Y. Katayama, K. Wada, N. Takahashi, M. Kobayashi, and M. Morita, “Self-stabilizing algorithms for constructing clusters and communication paths on WANET,” IPSJ Technical Report, 2007-AL-111, 2007.

[21] Y. Nakaminami, H. Kakugawa, and T. Masuzawa, “An advanced performance analysis of self-stabilizing protocols: Stabilization time with transient faults during convergence,” Proc. 8th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM), Paper number 106, 2006.

[22] G. Nishikawa, Y. Yamauchi, F. Ooshita, H. Kakugawa, and T. Masuzawa, “A fair self-stabilizing mutual exclusion protocol for mobile ad hoc networks,” IEICE Trans. Fundamentals (Japanese Edition), vol.J91-A, no.2, pp.279–284, Feb. 2008.

[23] H. Nishimura, T. Izumi, Y. Katayama, and K. Wada, “A self-stabilizing clustering algorithm based on maximal clique partition,” IPSJ Technical Report, 2008-AL-117, 2008.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, and IPSJ.