Gradual Stabilization under τ-Dynamics

Karine Altisen[1], Stéphane Devismes[1], Anaïs Durand[1], and Franck Petit[2]

[1] VERIMAG UMR 5104, Université Grenoble Alpes, France
[2] LIP6 UMR 7606, INRIA, UPMC Sorbonne Universités, France

Abstract. In this paper, we introduce the notion of *gradually stabilizing* algorithm as any self-stabilizing algorithm with the following additional feature: if at most τ *dynamic steps* occur starting from a legitimate configuration, it first quickly recovers to a configuration from which a minimum quality of service is satisfied and then gradually converges to stronger and stronger safety guarantees until reaching a legitimate configuration again. We illustrate this new property by proposing a gradually stabilizing unison algorithm.

1 Introduction

Self-stabilization [10] is a general paradigm to enable the design of distributed systems tolerating *any* finite number of transient faults. Consider the first configuration after all transient faults cease. This configuration is arbitrary, but no other transient faults will ever occur from this configuration. By abuse of language, this configuration is referred to as *arbitrary initial configuration* of the system in the literature. Then, a self-stabilizing algorithm (provided that faults have not corrupted its code) guarantees that starting from an arbitrary initial configuration, the system recovers *within finite time*, without any external intervention, to a so-called *legitimate configuration* from which its specification is satisfied. Thus, self-stabilization makes no hypotheses on the nature (*e.g.*, memory corruptions, topological changes) of transient faults, and the system recovers from the effects of those faults in a unified manner. Such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called *stabilization phase*, during which safety properties of the system may be violated. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, *i.e.*, the maximum duration of the stabilization phase. Many problem specifications induce a significant stabilization time, *e.g.*, in the context of synchronization tasks [3] and more generally for specifications of non-static problems [13], such as broadcast, the lower bound is $\Omega(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. By definition, the stabilization time is impacted by worst case scenarios, but, in many cases, transient faults are sparse and their effect may be superficial. Recent research thus focuses on proposing self-stabilizing algorithms that also ensure drastically smaller convergence times in favorable cases.

Defining the number of faults hitting a network using some kind of Hamming distance (minimal number of processes whose state must be changed in order to recover a legitimate configuration), variants of self-stabilization have been defined. A *time-adaptive* self-stabilizing algorithm [21] additionally guarantees a convergence time in

$O(k)$ time units when the initial configuration is at distance at most k from a legitimate configuration. *Fault containing* self-stabilizing algorithms [14] ensure that when few faults hit the system, the faults are both spatially and temporally contained. “Spatially” means that those faults cannot be propagated further than a preset radius around the corrupted processes. “Temporally” means quick stabilization when few faults occur. Some other approaches consist in providing convergence times *tailored by the type of transient faults*, *e.g.*, a *superstabilizing algorithm* [11] is self-stabilizing and has two additional properties when transient faults are limited to a *single* topological change: after adding or removing one link or process in the network, it recovers fast (typically $O(1)$ rounds), and a safety predicate, so-called *passage*, should be satisfied meanwhile.

Contributions. We introduce the notion of *gradually stabilizing* algorithm as any *self-stabilizing* algorithm achieving the following additional feature. If at most τ *dynamic steps*[3] occur starting from a legitimate configuration, a gradually stabilizing algorithm first quickly recovers to a configuration from which a specification offering a minimum quality of service is satisfied. It then gradually converges to specifications offering stronger and stronger safety guarantees until reaching a configuration from which its initial (strong) specification is satisfied again, and where it is ready to achieve another gradual convergence in case of up to τ new dynamic steps. Of course, this property makes sense only if convergence to every intermediate weaker specification is fast.

We illustrate this new property by considering three variants of a synchronization problem respectively called *strong*, *weak*, and *partial* (asynchronous) unison. In these problems, each process maintains a local clock. We restrict our study to periodic clocks, *i.e.*, clocks are integer variables whose domain is $\{0, \ldots, \alpha - 1\}$, where $\alpha \geq 2$ is called the *period*. Each process should regularly increment its clock modulo α (liveness) while fulfilling some safety requirements. The safety of *strong unison* requires that at most two consecutive clock values exist in each configuration of the system. *Weak unison* only requires that the difference between clocks of every two neighbors is at most one increment. Finally, we define *partial unison* as a specification dedicated to dynamic systems which enforces the difference between clocks to remain at most one increment, but only for neighboring processes that do not appear during the dynamic steps.

We propose a self-stabilizing strong unison algorithm which works with any period $\alpha > 4$ in any anonymous connected network. It assumes the knowledge of two values μ and β, where μ is any upper bound on n — the (initial) number of processes, α should divide β, and $\beta > \mu^2$. Our algorithm is designed in the locally shared memory model and assumes the distributed unfair daemon, the most general daemon of the model. Its stabilization time is at most $n + (\mu + 1)\mathcal{D} + 1$ rounds, where $\mathcal{D}$ is the diameter of the network. We then slightly modify this algorithm to make it gradually stabilizing after one dynamic step. In particular, the parameter μ should be now at least $n + \#J$, where $\#J$ is an upper bound on the number of processes that join the system during a dynamic step. This new version is *gradually stabilizing* because after one dynamic step from a configuration which is legitimate for strong unison, it immediately satisfies the specification of partial unison, then converges to the specification of weak unison in at most one round, and finally retrieves, after at most $(\mu + 1)\mathcal{D}_1 + 1$ additional rounds (where $\mathcal{D}_1$ is the diameter of the network after the dynamic step), a configuration from

[3] *N.b.*, a dynamic step is a step containing topological changes.

which the specification of strong unison is satisfied and where it is ready to achieve gradual convergence again in case of another dynamic step. This result holds considering dynamic steps which may contain several link and/or process additions and/or removals, however we assume that after a dynamic step, the network stays connected and, if $\alpha > 4$, every new process is linked to at least one process already in the system before the dynamic step. We show that this condition, called $\mathsf{UnderLocalControl}$, is necessary to obtain gradual convergence. However, notice that if the system suffers from arbitrary other kinds of transient fault including, *e.g.*, several dynamic steps that do not satisfy the $\mathsf{UnderLocalControl}$ condition, our algorithm still converges to strong unison, yet without intermediate safety guarantees during the stabilization phase.

Related Work. Gradual stabilization is related to two other stronger forms of self-stabilization: *safe-converging self-stabilization* [19] and *superstabilization* [11]. The goal of a safely converging self-stabilizing algorithm is to first quickly ($O(1)$ rounds is the usual rule) converge from an arbitrary configuration to a *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Hence, the aim of safe-converging self-stabilization is also to ensure a gradual convergence, but only for two specifications. However, such a gradual convergence is stronger than ours as it should be ensured after any step of transient faults,[4] while our gradual convergence applies after dynamic steps only. Safe convergence is especially interesting for self-stabilizing algorithms that compute optimized data structures, *e.g.*, minimal dominating sets [19], minimal (f, g)-alliances [8]. However, to the best of our knowledge, no safe-converging algorithm for non-static problems, such as unison, has been proposed until now.

In superstabilization, like in our approach, fast convergence and the passage predicate should be ensured only if the system was in a legitimate configuration before the topological change occurs. In contrast with our approach, superstabilization ensures fast convergence to the original specification. However, this strong property only considers one dynamic step with only *one* topological event. Again, superstabilization has been especially studied in the context of static problems, *e.g.*, spanning tree construction [4, 5, 11], and coloring [11]. However, there exist few superstabilizing algorithms for non-static problems in particular topologies, *e.g.*, mutual exclusion in rings [16, 20].

We use the general term *unison* to name several close problems also known in the literature as *phase* or *barrier synchronization* problems. There exist many self-stabilizing algorithms for strong or weak unison problems, *e.g.*, [2, 6, 7, 15, 17, 18, 22, 23]. However, to the best of our knowledge, until now there was no self-stabilizing solution for such problems addressing specific convergence properties in case of topological changes, in particular no superstabilizing one. Self-stabilizing strong unison was first considered in synchronous anonymous networks. Particular topologies were considered in [17] (rings) and [22] (trees). Gouda and Herman [15] proposed a self-stabilizing algorithm for strong unison working in anonymous synchronous systems of arbitrary connected topology. However, they considered unbounded clocks. A solution working with the same settings, yet implementing bounded clocks, is proposed in [2]. In [23], an asyn-

[4] Such transient faults may include topological changes, but not only.

chronous self-stabilizing strong unison algorithm is proposed for arbitrary connected rooted networks.

Johnen *et al* investigated asynchronous self-stabilizing weak unison in oriented trees in [18]. The first self-stabilizing asynchronous weak unison for general graphs was proposed by Couvreur *et al.* [9]. However, no complexity analysis was given. Another solution which stabilizes in $O(n)$ rounds has been proposed by Boulinier *et al.* in [7]. Finally, Boulinier proposed in his PhD thesis a parametric solution which generalizes both the solutions of [9] and [7]. In particular, the complexity analysis of this latter algorithm reveals an upper bound in $O(\mathcal{D}.n)$ rounds on the stabilization time of the Couvreur *et al.*' algorithm.

Roadmap. In the next section, we define the computational model used in this paper. In Section 3, we recall the formal definition of self-stabilization, and introduce the notion of gradual stabilization. In Section 4, we show that condition UnderLocalControl is necessary to obtain a gradually stabilizing solution. We present our self-stabilizing strong unison algorithm in Section 5. The gradually stabilizing variant of this latter algorithm is proposed in Section 6. We make concluding remarks in Section 7.

Due to the lack of space, proofs are omitted, see the report online [1] for details.

2 Preliminaries

We consider distributed systems made of *anonymous* processes. The system *initially contains $n > 0$ processes and its topology is connected*, however it may suffer from topological changes over time. Each process p can directly communicate with a subset $p.\mathcal{N}$ of other processes, its *neighbors*. In our context, $p.\mathcal{N}$ can vary over time. Communications are assumed to be *bidirectional* and carried out by a finite set of locally shared variables: each process can read its own variables and those of its current neighbors, but can only write into its own variables. The *state* of a process is the vector of values of its variables. We denote by $\mathcal{S}$ the set of all possible states of a process. Each process updates its variables according to a *local algorithm*. The collection of all local algorithms defines a *distributed algorithm*. The local algorithm of p consists of a finite set of *actions* of the following form: $\langle$ label $\rangle$ :: $\langle$ guard $\rangle \rightarrow \langle$ statement $\rangle$. *Labels* are used to identify actions in the reasoning. The *guard* of an action is a Boolean predicate involving variables of p and its neighbors. The *statement* is a sequence of assignments on variables of p. If the guard of some action evaluates to true, the action is said to be *enabled* at p. By extension, if at least one action is enabled at p, p is said to be enabled. An action can be executed only if it is enabled. The execution of an action consists in executing its statement, atomically. A *configuration* γ_i is a pair $(G_i, V_i \rightarrow \mathcal{S})$. $G_i = (V_i, E_i)$ is a simple undirected graph, where V_i is the set of processes that exist in γ_i and E_i represents the links between processes in γ_i. $V_i \rightarrow \mathcal{S}$ is a function which associates a state to any process of V_i. We denote by $\mathcal{C}$ the set of all possible configurations.

Executions. The dynamicity and asynchronism of the system are materialized by an adversary, called *daemon*. To perform a *step* from a configuration γ_i, the daemon can (1) activate processes that are enabled in γ_i — each activated process executes one of its enabled actions according to its state and that of its neighbors in γ_i, and/or (2) modify the topology. Activation of enabled processes and/or topology modifications are done atomically, leading to a new configuration γ_{i+1}. The set of all possible steps induces a binary relation $\mapsto$ over configurations (empty steps of the form $\gamma_i \mapsto \gamma_i$ are excluded). Relation $\mapsto$ is partitioned into $\mapsto_s$ and $\mapsto_d$. Relation $\mapsto_s$ defines all possible *static steps* consisting in activation of enabled processes *only*. Relation $\mapsto_d$ defines all possible *dynamic steps* containing topological changes and possibly process activations.

An *execution* is a sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that G_0 is connected and $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$. For sake of simplicity, we note $G_0 = G = (V, E)$; we also note $\mathcal{D}$ the diameter of G. Moreover, we note $\mathcal{E}^\tau$ the set of maximal executions which contain at most τ dynamic steps. The set of all possible executions is therefore equal to $\mathcal{E} = \cup_{\tau \geq 0} \mathcal{E}^\tau$. For any subset of configurations $X \subseteq \mathcal{C}$, we denote by $\mathcal{E}^\tau_X$ the set of all executions in $\mathcal{E}^\tau$ that start from a configuration of X.

Dynamic Steps. Any step $\gamma_i \mapsto_d \gamma_{i+1}$ contains a finite number of topological events and maybe some process activations. Each topological event is of the following types. (1) A process p can *join* the system. This event, denoted by $join_p$, triggers the atomic execution of a specific action, called *bootstrap*. This bootstrap is executed without any communication and initializes the variables of p to a particular state, called *bootstate*. We denote by New_k the set of processes which are in bootstate in γ_k. When p joins the system in $\gamma_i \mapsto_d \gamma_{i+1}$, we have $p \in New_{i+1}$, but $p \notin New_i$. Until p executes its bootstrap, say in step $\gamma_x \mapsto \gamma_{x+1}$, it is still in bootstate. Hence, $\forall j \in \{i{+}1, \ldots, x\}, p \in New_j$, but $p \notin New_{x+1}$. We assume that there are at most $\#J$ joins during a dynamic step. (2) A process can also *leave* the system. (3) Finally, some communication links can *appear* or *disappear* between two different processes.

Daemon. We assume the daemon is *distributed* and *unfair*. In a static step, this daemon must select at least one enabled process. In a dynamic step, it can select zero, one, or several enabled processes. It has no fairness constraint, *i.e.*, it might never select a process p during any step unless in the case of a static step from a configuration where p is the only enabled process. Moreover, at each configuration, it freely chooses between making a static or dynamic step, except if no more process is enabled; in this latter case, only a dynamic step containing no process activation can be chosen.

Metrics. We measure the time complexity of our algorithms in *rounds* [12]. The first round of an execution $e = (\gamma_i)_{i \geq 0}$ is the minimal prefix e' of e in which every process that is enabled in γ_0 either disappears, or executes an action, or becomes disabled (due to some changes in its neighborhood). Let γ_j be the last configuration of e', the second round of e is the first round of $e'' = (\gamma_i)_{i \geq j}$, and so on.

Specifications. We define a specification as a predicate over executions. We denote by SP_{SU} and SP_{WU} the respective specifications for *strong* and *weak* unison. The specification of *partial* unison, noted SP_{PU}, does not impose any constraint on processes that join the system until they achieve their bootstrap: the safety holds as long as clocks of every two neighboring processes *not in bootstate* differ from at most one increment.

3 Stabilization

Self-stabilization has been defined by only considering executions free of topological changes, yet starting from an arbitrary configuration. Indeed, self-stabilization considers the system immediately after transient faults cease. So, the system is initially observed from an arbitrary configuration reached after occurrences of transient faults (including some topological changes maybe), but from which no faults will ever occur. Below, we recall the definitions of some notions classically used in self-stabilization for a given distributed algorithm $\mathcal{A}$. Let X and Y be two subsets of configurations.

- X is *closed under $\mathcal{A}$ iff* every *static step* of $\mathcal{A}$ starting from a configuration of X leads to a configuration which is also in X.
- *Y converges to X under $\mathcal{A}$ iff* every execution of $\mathcal{E}_Y^0$ contains a configuration of X.
- *$\mathcal{A}$ stabilizes from Y to a specification SP by X iff* X is closed under $\mathcal{A}$, Y converges to X under $\mathcal{A}$, and every execution of $\mathcal{E}_X^0$ satisfies SP. In this case, the *convergence time from Y to X in rounds* is the maximal number of rounds to reach a configuration of X over every execution of $\mathcal{E}_Y^0$.

A distributed algorithm $\mathcal{A}$ is *self-stabilizing* for a specification SP *iff* $\exists \mathcal{L} \subseteq \mathcal{C}$ such that $\mathcal{A}$ stabilizes from $\mathcal{C}$ to SP by $\mathcal{L}$. $\mathcal{L}$ is said to be a set of *legitimate configurations w.r.t.* SP, and the convergence time from $\mathcal{C}$ to $\mathcal{L}$ is called *stabilization time* of $\mathcal{A}$.

Gradual Stabilization Under τ-Dynamics. This property is a specialization of self-stabilization which additionally requires that after at most τ dynamic steps from a legitimate configuration, the system gradually re-stabilizes to stronger and stronger specifications, until fully recovering its initial (strong) specification. For every execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}^\tau$, we note $\gamma_{fst(e)}$ the first configuration of e after the last dynamic step. Formally, $fst(e) = \min\{i : (\gamma_j)_{j \geq i} \in \mathcal{E}^0\}$. For any subset E of $\mathcal{E}^\tau$, let $FC(E) = \{\gamma_{fst(e)} : e \in E\}$ be the set of all configurations that can be reached after the last topological changes in executions of E. Let $SP_1, SP_2, \ldots, SP_k$, be an ordered sequence of specifications. Let $B_1, B_2, \ldots, B_k$ be (asymptotic) complexity bounds such that $B_1 \leq B_2 \leq \cdots \leq B_k$.

A distributed algorithm $\mathcal{A}$ is *gradually stabilizing under τ-dynamics* for $(SP_1 \bullet B_1, SP_2 \bullet B_2, \ldots, SP_k \bullet B_k)$ *iff* $\exists \mathcal{L}_1, \ldots, \mathcal{L}_k \subseteq \mathcal{C}$ such that

1. $\mathcal{A}$ stabilizes from $\mathcal{C}$ to SP_k by $\mathcal{L}_k$.
2. $\forall i \in \{1, \ldots, k\}$, $\mathcal{A}$ stabilizes from $FC(\mathcal{E}^\tau_{\mathcal{L}_k})$ to SP_i by $\mathcal{L}_i$, and the convergence time in rounds from $FC(\mathcal{E}^\tau_{\mathcal{L}_k})$ to $\mathcal{L}_i$ is bounded by B_i.

The first point ensures that a gradually stabilizing algorithm is still self-stabilizing for its strongest specification. Hence, its performances can be also evaluated at the light of its stabilization time. Indeed, it captures the maximal convergence time of the gradually stabilizing algorithm after the system suffers from an arbitrary finite number of transient faults, *e.g.*, after more than τ dynamic steps.

The second point means that after at most τ dynamic steps from a legitimate configuration *w.r.t.* the strongest specification SP_k, the algorithm *gradually converges* to every specification SP_i with $i \in \{1, \ldots, k\}$ in at most B_i rounds. Note that B_k captures a complexity similar to the *fault gap* in fault-containing algorithms [14]: assume a period of at most τ dynamic steps starting in a legitimate configuration $\mathcal{L}_k$; B_k represents the necessary fault-free interval after this period and before the next period of at most τ dynamic steps so that system becomes ready to achieve gradual convergence again.

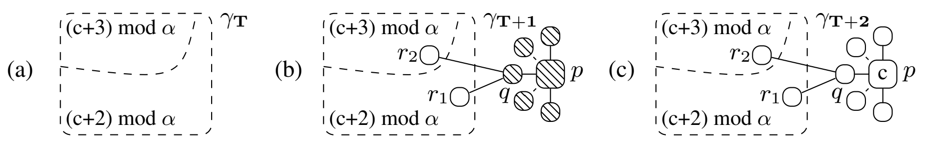

Fig. 1: Proof outline of Theorem 1. The hachured nodes are in bootstate.

4 Necessary Condition

In this section, we establish that Condition UnderLocalControl is necessary to allow the design of a deterministic algorithm $\mathcal{A}$ which is gradually stabilizing under 1-dynamics for $(SP_{\mathsf{PU}} \bullet 0, SP_{\mathsf{WU}} \bullet 1, SP_{\mathsf{SU}} \bullet B)$ (with $B \geq 1$) in any arbitrary anonymous network, assuming the distributed unfair daemon. Below, we assume the existence of $\mathcal{A}$ and denote by $\mathcal{L}^{\mathcal{A}}_{\mathsf{SU}}$ the set of legitimate configurations of $\mathcal{A}$ *w.r.t.* specification SP_{SU}.

UnderLocalControl captures a condition on the network dynamics which is necessary to prevent a notable desynchronization of clocks: the network should stay connected and, if $\alpha > 4$, every process that joins during the dynamic step $\gamma \mapsto_d \gamma'$ should be "under control of" (that is, linked to) at least one process which exists in both γ and γ'. The definition of UnderLocalControl uses the notion of *dominating set* of a graph $G = (V, E)$, *i.e.*, any subset D of V such that every node not in D is adjacent to at least one member of D. Formally, UnderLocalControl holds *iff* $\forall e \in \mathcal{E}^1_{\mathcal{L}^{\mathcal{A}}_{\mathsf{SU}}}$, $G_{fst(e)}$ is connected, and if $\alpha > 4$, then $V_{fst(e)} \setminus New_{fst(e)}$ is a dominating set of $G_{fst(e)}$.

Theorem 1. *An algorithm $\mathcal{A}$ is gradually stabilizing under 1-dynamics for $(SP_{\mathsf{PU}} \bullet 0, SP_{\mathsf{WU}} \bullet 1, SP_{\mathsf{SU}} \bullet B)$ in arbitrary anonymous networks under the distributed unfair daemon only if UnderLocalControl holds.*

Proof Outline. If the graph becomes disconnected after a dynamic step, the distributed unfair daemon can prevent forever all processes of a given connected component from incrementing their clocks, hence violating the liveness of SP_{SU}. Assume, by contradiction, that there is an execution e with $\alpha > 4$ such that $G_{fst(e)}$ is connected but $V_{fst(e)} \setminus New_{fst(e)}$ is not a dominating set. This means that some process p and all its neighbors have been added during the dynamic step. First, to satisfy SP_{WU} after at most one round, p and its neighbors should be enabled to take a clock value immediately after the dynamic step. Let c be the clock value that p would choose in this case. Then, we build another execution e' initiated from a configuration in $\mathcal{L}^{\mathcal{A}}_{\mathsf{SU}}$ on another graph containing at least two nodes which are neither p, nor one of its neighbors. As SP_{SU} holds and the execution can be asynchronous, it is possible for the system to eventually reach a configuration γ_T where there are exactly two clock values: $(c + 2) \mod \alpha$ and $(c + 3) \mod \alpha$ (see Fig. 1(a)). Then, assume the daemon chooses to execute, during $\gamma_T \mapsto_d \gamma_{T+1}$, the dynamic step which contains no process activation, but introduces p, its neighborhood, and two links, just as in Fig. 1(b). Then, after this step, SP_{PU} should be satisfied. Finally, assume that the daemon selects no process, except p and its neighbors in the next step. As before, p sets its clock to c, but, as $\alpha > 4$, whatever be the value chosen by q, there is a difference greater than one increment between q and at least one of its neighbors (Fig. 1(c)). Henceforth, the legitimate configurations of SP_{PU} are not closed under $\mathcal{A}$, a contradiction. □

5 Self-Stabilizing Strong Unison

In this section, we propose an algorithm which is self-stabilizing for strong unison in any arbitrary connected anonymous network. This algorithm works for any period $\alpha > 4$ and is based on an algorithm previously proposed by Boulinier in his PhD [6], this latter is self-stabilizing for weak unison and works for any period $\beta > n^2$.

Algorithm $\mathcal{WU}$. We first recall the algorithm of Boulinier [6], noted here Algorithm $\mathcal{WU}$. This algorithm being just self-stabilizing, it only considers executions without any topological change, yet starting from arbitrary configurations. So, the topology of the network consists in a connected graph $G = (V, E)$ of n nodes which is fixed all along the execution. Each process p is endowed with a clock variable $p.t \in \{0, \ldots, \beta - 1\}$, where β is its period. β should be greater than n^2. The algorithm also uses another constant, noted μ, which should satisfy $n \leq \mu \leq \frac{\beta}{2}$. The algorithm uses the notion of *delay* between two integer values x and y, defined by the function $d_\beta\big(x, y\big) = \min\big((x - y) \mod \beta, (y - x) \mod \beta\big)$. It also uses the relation $\preceq_{\beta,\mu}$ such that for every two integer values x and y, $x \preceq_{\beta,\mu} y \equiv \big((y - x) \mod \beta\big) \leq \mu$.

Two actions are used to maintain the clock $p.t$ at each process p. When the delay between $p.t$ and the clocks of some neighbors is greater than one, but the maximum delay is not too big (that is, does not exceed μ), then it is possible to "normally" converge, using Action $\mathcal{WU}$-N below, to a configuration where the delay between those clocks is at most one by incrementing the clocks of the most behind processes among p and its neighbors: $\mathcal{WU}\text{-}N :: \forall q \in p.\mathcal{N}, p.t \preceq_{\beta,\mu} q.t \rightarrow p.t \leftarrow (p.t + 1) \mod \beta$

Moreover, once legitimacy is achieved, p can "normally" increment its clock still using Action $\mathcal{WU}$-N when it is on time or one increment late with all its neighbors. In contrast, if the delay is too big (that is, the delay between the clocks of p and one of its neighbors is more than μ) and the clock of p is not yet reset, then p should reset its clock to 0 using Action $\mathcal{WU}$-R: $\mathcal{WU}\text{-}R :: \exists q \in p.\mathcal{N}, d_\beta\big(p.t, q.t\big) > \mu \wedge p.t \neq 0 \rightarrow p.t \leftarrow 0$

Algorithm $\mathcal{SU}$. For this algorithm, we still assume a non-dynamic context (no topological change). Algorithm $\mathcal{SU}$ is a straightforward adaptation of Algorithm $\mathcal{WU}$. More precisely, Algorithm $\mathcal{SU}$ maintains two clocks at each process p. The first one, $p.t \in \{0, \ldots, \beta - 1\}$, is called the *internal clock* and is maintained exactly as in Algorithm $\mathcal{WU}$. Then, $p.t$ is used as an internal pulse machine to increment a second, yet actual, clock of Algorithm $\mathcal{SU}$ $p.c \in \{0, \ldots, \alpha - 1\}$, also called *external clock*.

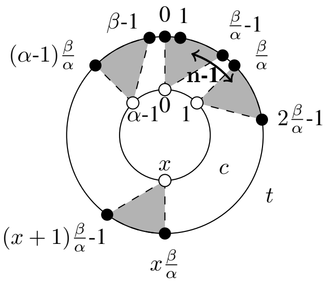

Fig. 2: From t to c.

Algorithm $\mathcal{SU}$ is designed for any period $\alpha > 4$. Its actions $\mathcal{SU}$-N and $\mathcal{SU}$-R are identical to actions $\mathcal{WU}$-N and $\mathcal{WU}$-R of Algorithm $\mathcal{WU}$, except that we add the computation of the external c-clock in their respective statement.

$\mathcal{SU}$-N :: $\forall q \in p.\mathcal{N}, p.t \preceq_{\beta,\mu} q.t \qquad\qquad \rightarrow p.t \leftarrow (p.t+1) \mod \beta;\ p.c \leftarrow \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$

$\mathcal{SU}$-R :: $\exists q \in p.\mathcal{N}, d_\beta(p.t, q.t) > \mu \wedge p.t \neq 0 \rightarrow p.t \leftarrow 0;\ p.c \leftarrow 0$

Algorithm $\mathcal{WU}$ stabilizes to a configuration from which t-clocks regularly increment while preserving a bounded delay of at most one between two neighboring processes, and so of at most $n-1$ between any two processes. Algorithm $\mathcal{SU}$ implements the same mechanism to maintain $p.t$ at each process p and computes $p.c$ from $p.t$ as a normalization operation from clock values in $\{0, \ldots, \beta-1\}$ to $\{0, \ldots, \alpha-1\}$: each time the value of $p.t$ is modified, $p.c$ is updated to $\left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$. Hence, we can set β in such way that $K = \frac{\beta}{\alpha}$ is greater than or equal to n (here, we choose $K > \mu \geq n$ and $\beta > \mu^2$ for sake of simplicity) to ensure that, when the delay between any two t-clocks is at most $n-1$, the delay between any two c-clocks is at most one, see Fig. 2. The liveness of $\mathcal{WU}$ ensures that every t-clock increments infinitely often, thus so do c-clocks.

Theorem 2. *Algorithm $\mathcal{SU}$ is self-stabilizing for SP_{SU} in any arbitrary connected anonymous network assuming a distributed unfair daemon. Its stabilization time is at most $n + (\mu+1)\mathcal{D} + 1$ rounds.*

We have also proven that, once $\mathcal{SU}$ has stabilized, every process increments its c-clock at least once every $\mathcal{D} + \frac{\beta}{\alpha}$ rounds. This result derives from [6] which states that after stabilization of t-clocks, those ones increment at least once every $\mathcal{D}+1$ rounds.

6 Gradual Stabilization under 1-Dynamics for Strong Unison

We now propose Algorithm $\mathcal{DSU}$ (Algorithm 1), a variant of Algorithm $\mathcal{SU}$. $\mathcal{DSU}$ is still self-stabilizing for strong unison, but also achieves a gradual convergence after one dynamic step. This dynamic step may include several topological events (*i.e.* link or process additions or removals). However, according to Theorem 1, it should satisfy Condition UnderLocalControl. Precisely, after any dynamic step which fulfills condition UnderLocalControl, $\mathcal{DSU}$ maintains clocks almost synchronized during the convergence to strong unison since it immediately satisfies partial unison, then converges in at most one round to weak unison, and finally re-stabilizes to strong unison. Remember that, after one dynamic step, the graph contains at most $n + \#J$ processes, by definition, and $\mathcal{D}_1$ denotes the diameter of the new graph.

We first showed a result allowing to simplify proofs and explanations: for every closed set of configurations X, if UnderLocalControl holds, then $\forall \gamma_i \in \mathcal{C}, (\exists \gamma_j \in X \mid \gamma_j \mapsto_d \gamma_i) \Leftrightarrow (\exists \gamma_k \in X \mid \gamma_k \mapsto_{d_{only}} \gamma_i)$, where $\mapsto_{d_{only}}$ is the relation defining all dynamic steps containing no process activation. We apply this result to the set of legitimate configurations *w.r.t.* strong unison, noted $\mathcal{L}^d_{\mathsf{SU}}$ (*n.b.*, $\mathcal{L}^d_{\mathsf{SU}}$ is closed, by definition): the set of configurations reachable from $\mathcal{L}^d_{\mathsf{SU}}$ after one dynamic step (which may also include process activations) is the same as the one reachable from $\mathcal{L}^d_{\mathsf{SU}}$ after one dynamic step made of topological events only. At the light of this result, we only consider this latter kind of dynamic steps in the following.

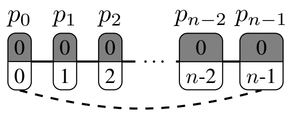

Fig. 3: Link addition.

Algorithm 1 $\mathcal{DSU}$, for every process p

```
Parameters:
    $\alpha$: any positive integer such that $\alpha > 4$
    $\mu$: any positive integer such that $\mu \geq n + \#J$
    $\beta$: any positive integer such that $\beta > \mu^2$, and $\exists K$ such that $K > \mu$ and $\beta = K\alpha$
Variables: $p.c \in \{0, \ldots, \alpha - 1\} \cup \{\bot\}$,  $p.t \in \{0, \ldots, \beta - 1\} \cup \{\bot\}$
Predicates:
    $Locked_p \equiv p.t = \bot \vee \exists q \in p.\mathcal{N}, q.t = \bot$
    $NormalStep_p \equiv \neg Locked_p \wedge \forall q \in p.\mathcal{N}, p.t \preceq_{\beta,\mu} q.t$
    $ResetStep_p \equiv \neg Locked_p \wedge \left(\exists q \in p.\mathcal{N}, d_\beta\left(p.t, q.t\right) > \mu \wedge\ p.t \neq 0\right)$
    $JoinStep_p \equiv p.t = \bot$
Actions:
    $\mathcal{DSU}\text{-}N$ :: $NormalStep_p \to p.t \leftarrow (p.t + 1) \bmod \beta;\ p.c \leftarrow \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$
    $\mathcal{DSU}\text{-}R$ :: $ResetStep_p \to p.t \leftarrow 0;\ p.c \leftarrow 0$
    $\mathcal{DSU}\text{-}J$ :: $JoinStep_p \to p.t \leftarrow MinTime_p;\ p.c \leftarrow \left\lfloor \frac{\alpha}{\beta} p.t \right\rfloor$
    $bootstrap$ :: $join_p \to p.t \leftarrow \bot;\ p.c \leftarrow \bot$
```

Consider first link additions only. Adding a link (see the dashed link in Fig. 3) can break the safety of weak unison on internal clocks. Indeed, it may create a delay greater than one between two new neighboring t-clocks. Nevertheless, the delay between any two t-clocks remains bounded by $n - 1$, consequently, no process will reset its t-clock (Fig. 3 shows a worst case). Moreover, c-clocks still satisfy strong unison immediately after the link addition. Besides, since increments are constrained by neighboring clocks, adding links only reinforces those constraints. Thus, the delay between internal clocks of arbitrary far processes remains bounded by $n - 1$, and so strong unison remains satisfied, in all subsequent static steps. Consider again the example in Fig. 3: before the dynamic step, p_{n-1} had only to wait until p_{n-2} increments $p_{n-2}.t$ in order to be able to increment its own t-clock; yet after the step, it also has to wait for p_0.

Assume now a dynamic step containing only process and link removals. Due to Condition UnderLocalControl, the network remains connected. Hence, constraints between (still existing) neighbors are maintained: the delay between t-clocks of two neighbors remains bounded by one, see the example in Fig. 4: process p_2 and link $\{p_0, p_3\}$ are removed. So, weak unison on t-clocks remains satisfied and so is strong unison on c-clocks.

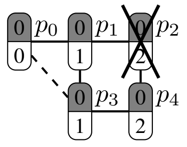

Fig. 4: Removals.

Consider now a more complex scenario, where the dynamic step contains link additions as well as process and/or link removals. Fig. 5 shows an example of such a scenario, where safety of strong unison is violated. As above, the addition of link $\{p_1, p_6\}$ in Fig. 5(b) leads to a delay between t-clocks of these two (new) neighbors which is greater than one (here 5). However, the removal of link $\{p_1, p_2\}$, also in Fig. 5(b), relaxes the neighborhood constraint on p_2: p_2 can now increment without waiting for p_1. Consequently, executing Algorithm $\mathcal{SU}$ does not ensure that the delay between t-clocks of any two arbitrary far processes remains bounded by $n - 1$, *e.g.*, after several static steps from Fig. 5(b), the system can reach Fig. 5(c), where the delay between p_1 and p_2 is 9 while $n - 1 = 5$. Since c-clock values are computed from t-clock values, we also cannot guarantee that there is at most two consecutive c-clock values in the system, *e.g.*, in Fig. 5(c) we have: $p_1.c = 1$, $p_6.c = 2$, and $p_2.c = 3$.

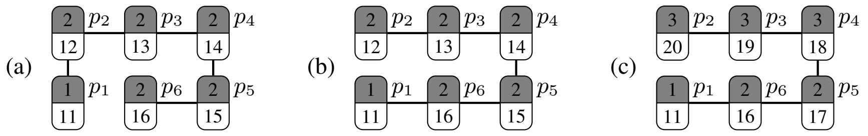

Fig. 5: Execution where links are added and removed ($\mu = 6$, $\alpha = 7$, and $\beta = 42$).

Again, in the worst case scenario, after a dynamic step, the delay between two neighboring t-clocks is bounded by $n - 1$. Moreover, t-clocks being computed like in Algorithm $\mathcal{WU}$, we can use two of its useful properties (see [6]): (1) when the delay between every pair of neighboring t-clocks is at most μ with $\mu \geq n$, the delay between these clocks remains bounded by μ because processes never reset; (2) furthermore, from such configurations, the system converges to a configuration from which the delay between the t-clocks of every two neighbors is at most one. So, keeping $\mu \geq n$, processes will not reset after one dynamic step and the delay between any two neighboring t-clocks will monotonically decrease from at most $n-1$ to at most one. Consequently, the delay between any two neighboring c-clocks (which are computed from t-clocks) will stay at most one, *i.e.*, weak unison will be satisfied all along the convergence to strong unison.

Consider now a process p that joins the system. The event $join_p$ occurs and triggers the specific action $bootstrap$ that sets both the clocks $p.t$ and $p.c$ to a specific *bootstate* value, noted $\perp$. By definition and from the previous discussion, the system immediately satisfies partial unison since it only depends on processes that were in the system before the dynamic step. Now, to ensure that weak unison holds within a round, we add the action $\mathcal{DSU}$-J which is enabled as soon as the process is in bootstate. This action initializes the two clocks of p according to the clock values in its neighborhood. Precisely, the value of $p.t$ can be chosen among the non-$\perp$ values in its neighborhood, and such values exist by Condition UnderLocalControl. We choose to set $p.t$ to the minimum non-$\perp$ t-clock value in its neighborhood, using the function $MinTime_p$:
$MinTime_p = 0$ **if** $\forall q \in p.\mathcal{N}, q.t = \perp$**;** $= \min\{q.t : q \in p.\mathcal{N} \wedge q.t \neq \perp\}$ **otherwise.**
The value of $p.c$ is then computed according to the value of $p.t$. Notice that $MinTime_p$ returns 0 when p and all its neighbors have their respective t-clock equal to $\perp$. This ensures that Algorithm $\mathcal{DSU}$ remains self-stabilizing (in particular, if the system starts in a configuration where all t-clocks are equal to $\perp$).

To prevent the unfair daemon from blocking the convergence to a configuration containing no $\perp$ values, we should also forbid processes with non-$\perp$ t-clock values to increment while there are t-clocks with $\perp$-values in their neighborhood. So, we define the predicate $Locked$ which holds for a given process p when either $p.t = \perp$, or at least one of its neighbors q satisfies $q.t = \perp$. We then enforce the guard of both normal and reset actions, so that no $Locked$ process can execute them. See actions $\mathcal{DSU}$-N and $\mathcal{DSU}$-R. This ensures that t-clocks are initialized first by Action $\mathcal{DSU}$-J, before any value in their neighborhood increments.

Finally, notice that all the previous explanation relies on the fact that, once the system recovers from process additions (*i.e.*, once no $\perp$ value remains), the algorithm behaves exactly the same as Algorithm $\mathcal{SU}$. Hence, it has to match the assumptions made for $\mathcal{SU}$, in particular, the ones on α and β. However the constraint on μ has to

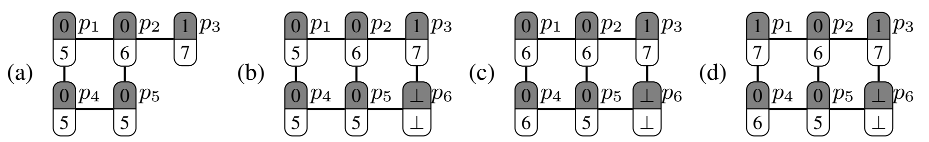

Fig. 6: Execution where the first step of a new process is delayed ($\mu = 6$, $\alpha = 6$, $\beta = 42$).

be adapted, since μ should be greater than or equal to the actual number of processes in the network and this number may increase. Now, the number of processes added in a dynamic step is bounded by $\#J$. So, we require $\mu \geq n + \#J$.

We now consider the example execution of Algorithm $\mathcal{DSU}$ in Fig. 6. This execution starts in a configuration legitimate *w.r.t.* the strong unison, see Fig. 6(a). Then, one dynamic step happens (step (a)$\mapsto$(b)), where a process p_6 joins the system. We now try to delay as long as possible the execution of $\mathcal{DSU}$-J by p_6. In configuration (b), p_3 and p_5, the new neighbors of p_6, are locked. They will remain disabled until p_6 executes $\mathcal{DSU}$-J. p_1 and p_4 execute $\mathcal{DSU}$-N in (b)$\mapsto$(c). Then, p_4 is disabled because of p_5 and p_1 executes $\mathcal{DSU}$-N in (c)$\mapsto$(d). In configuration (d), p_1 is from now on disabled: p_1 must wait until p_2 and p_4 get t-clock value 7. p_6 is the only enabled process, so the unfair daemon has no other choice but selecting p_6 to execute $\mathcal{DSU}$-J in the next step.

Theorem 3. *If UnderLocalControl is satisfied then Algorithm $\mathcal{DSU}$ is gradually stabilizing under 1-dynamics for* $(SP_{PU} \bullet 0, SP_{WU} \bullet 1, SP_{SU} \bullet (\mu + 1)\mathcal{D}_1 + 2)$.

After one dynamic step that fulfills Condition UnderLocalControl from any legitimate configuration *w.r.t.* strong unison, the system re-stabilizes to strong unison in at most $(\mu + 1)\mathcal{D}_1 + 2$ rounds. Now, in any other cases (*e.g.*, a dynamic step that does not satisfy UnderLocalControl), the system still recovers to a legitimate configuration within finite time, as the algorithm is self-stabilizing. Nevertheless, in such cases, the stabilization time is slightly bigger: $n + \#J + (\mu + 1)\mathcal{D}_1 + 2$ rounds.

Finally, we have proven [1] that after stabilization to strong unison, every process increments its c-clock at least once every $\mathcal{D}_1 + \frac{\beta}{\alpha}$ rounds, like in Algorithm $\mathcal{SU}$. Moreover, during the convergence from weak to strong unison, the increments are slower, *i.e.*, the c-clocks are guaranteed to increment at least once every $\mu\mathcal{D}_1 + \frac{\beta}{\alpha}$ rounds.

7 Conclusion

The apparent seldomness of superstabilizing solutions for non-static problems, such as unison, may suggest the difficulty of obtaining such a strong property and if so, make our notion of gradual stabilization very attractive compared to merely self-stabilizing solutions. For example, in our unison solution, gradual stabilization ensures that processes remain "almost" synchronized during the convergence phase started after one dynamic step satisfying UnderLocalControl. Hence, it is worth investigating whether this new paradigm can be applied to other, in particular non-static, problems. Concerning our unison algorithm, the graceful recovery after one dynamic step comes at the price of slowing down the clock increments. The question of limiting this drawback remains open. Finally, it would be interesting to address in future work gradual stabilization for non-static problems in context of more complex dynamic patterns.

References

1. Altisen, K., Devismes, S., Durand, A., Petit, F.: Gradual Stabilization under τ-Dynamics. Tech. rep. (2015), https://hal.archives-ouvertes.fr/hal-01215190
2. Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. Parallel Processing Letters 1, 11–18 (1991)
3. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: STOC. pp. 652–661 (1993)
4. Blin, L., Potop-Butucaru, M., Rovedakis, S.: A super-stabilizing log(n)log(n)-approximation algorithm for dynamic steiner trees. Theor. Comput. Sci. 500, 90–112 (2013)
5. Blin, L., Potop-Butucaru, M.G., Rovedakis, S., Tixeuil, S.: Loop-free super-stabilizing spanning tree construction. In: SSS. pp. 50–64 (2010)
6. Boulinier, C.: L'Unisson. Ph.D. thesis, Université de Picardie Jules Vernes, France (2007)
7. Boulinier, C., Petit, F., Villain, V.: When graph theory helps self-stabilization. In: PODC. pp. 150–159 (2004)
8. Carrier, F., Datta, A.K., Devismes, S., Larmore, L.L., Rivierre, Y.: Self-stabilizing (f,g)-alliances with safe convergence. J. Parallel Distrib. Comput. 81-82, 11–23 (2015)
9. Couvreur, J., Francez, N., Gouda, M.G.: Asynchronous unison (extended abstract). In: ICDCS. pp. 486–493 (1992)
10. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
11. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. 1997 (1997)
12. Dolev, S., Israeli, A., Moran, S.: Uniform Dynamic Self-Stabilizing Leader Election. IEEE Trans. Parallel Distrib. Syst. 8(4), 424–440 (1997)
13. Genolini, C., Tixeuil, S.: A lower bound on dynamic k-stabilization in asynchronous systems. In: SRDS. p. 212 (2002)
14. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Distributed Computing 20(1), 53–73 (2007)
15. Gouda, M.G., Herman, T.: Stabilizing unison. Inf. Process. Lett. 35(4), 171–175 (1990)
16. Herman, T.: Superstabilizing mutual exclusion. Distributed Computing 13(1), 1–17 (2000)
17. Huang, S., Liu, T.: Four-state stabilizing phase clock for unidirectional rings of odd size. Inf. Process. Lett. 65(6), 325–329 (1998)
18. Johnen, C., Alima, L.O., Datta, A.K., Tixeuil, S.: Optimal snap-stabilizing neighborhood synchronizer in tree networks. Parallel Processing Letters 12(3-4), 327–340 (2002)
19. Kakugawa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: IPDPS. pp. 8.– (2006)
20. Katayama, Y., Ueda, E., Fujiwara, H., Masuzawa, T.: A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. J. Parallel Distrib. Comput. 62(5), 865–884 (2002)
21. Kutten, S., Patt-Shamir, B.: Stabilizing time-adaptive protocols. Theor. Comput. Sci. 220(1), 93–111 (1999)
22. Nolot, F., Villain, V.: Universal self-stabilizing phase clock protocol with bounded memory. In: IPCCC. pp. 228–235 (2001)
23. Tzeng, C., Jiang, J., Huang, S.: Size-independent self-stabilizing asynchronous phase synchronization in general graphs. J. Inf. Sci. Eng. 26(4), 1307–1322 (2010)