Distrib. Comput. (2007) 20:53–73
DOI 10.1007/s00446-007-0032-2

Fault-containing self-stabilizing distributed protocols

Sukumar Ghosh · Arobinda Gupta · Ted Herman · Sriram V. Pemmaraju

Received: 16 July 1999 / Accepted: 5 July 2000 / Published online: 2 June 2007

Abstract Self-stabilization is an elegant approach for designing a class of fault-tolerant distributed protocols. A self-stabilizing protocol is guaranteed to eventually converge to a legitimate state after a transient fault. However, even a minor transient fault can cause vast disruption in the system before legitimacy is reached. This paper introduces the notion of fault-containment to address this particular weakness of self-stabilizing systems. Informally, a fault-containing self-stabilizing protocol, in addition to providing self-stabilization, *contains* the effects of faults. This ensures that disruption during recovery from faults, is proportional to the extent of the faults. The paper begins with a formal framework for specifying and evaluating fault-containing self-stabilizing protocols. The main result of the paper is a transformer that converts any non-reactive self-stabilizing protocol into an equivalent fault-containing self-stabilizing protocol that can repair any single fault in the system in $O(1)$ time. For a large class of input protocols, the corresponding output protocols produced by the transformer have $O(1)$ space overhead. The small time and space overhead make the fault-containing self-stabilizing protocol a practical alternative to the original self-stabilizing protocol. The transformer is based on a novel stabilizing timer paradigm that significantly simplifies the task of fault-containment.

S. Ghosh (✉) · T. Herman · S. V. Pemmaraju
Department of Computer Science, The University of Iowa,
Iowa City, IA 52242, USA
e-mail: ghosh@cs.uiowa.edu

T. Herman
e-mail: herman@cs.uiowa.edu

S. V. Pemmaraju
e-mail: sriram@cs.uiowa.edu

A. Gupta
Department of Computer Science and Engineering,
Indian Institute of Technology,
Kharagpur 721302, India
e-mail: agupta@cse.iitkgp.ernet.in

Keywords Distributed algorithms · Self-stabilization · Fault-containment · Transformer

1 Introduction

Self-stabilization is an elegant approach for designing a class of fault tolerant distributed protocols. A system is said to be self-stabilizing if starting from any initial state, the system is guaranteed to converge to, and stay in thereafter, a state belonging to a well-defined set of desirable states. The set of desirable states to which the system converges is called the set of *legitimate states*; the rest of the states are labeled *illegitimate states*. Research on self-stabilization was initiated by a seminal paper by Dijkstra [14] in 1974. Since then, self-stabilizing protocols have been designed for a large variety of problems and underlying principles have been explored [16,34].

Self-stabilizing systems are important in the area of fault-tolerance because they provide automatic tolerance to *transient faults*. Since the state of a system after transient faults can be viewed as an arbitrary initial state, the system automatically recovers from such faults and reaches a legitimate state without any external intervention. However, the importance of self-stabilizing systems is not limited to their tolerance of transient faults. In many cases, self-stabilizing protocols can dynamically adapt to changes in topology of the underlying network [3,12,15,18,23]. Therefore, such protocols can be thought of as being tolerant to permanent faults, such as the crash of a node or a link, that change the topology of the network. Some self-stabilizing protocols [10,23] have the ability to automatically adjust to dynamic changes in the parameters of a problem such as weights associated with edges or nodes in the network. Self-stabilizing protocols have also been designed in some cases to tolerate more severe permanent faults such as omission or Byzantine faults [11,25].

The motivation of our current research stems from a fundamental limitation of self-stabilizing systems. Fault-tolerance can be classified into two broad categories: *masking* and *non-masking* (see Arora and Kulkarni [4]). *Masking* fault-tolerance is the ability of programs to continually satisfy their specification in the presence of faults. Self-stabilization is considered *non-masking* fault-tolerance, since the users of a stabilizing system can observe disrupted behavior while the system recovers to a legitimate state. Given a non-masking fault-tolerant system, one would hope that the level of disruption observable by users be proportional to the severity of the fault causing the disruption. Unfortunately, many stabilizing systems do not have this property: in some cases even a single bit corruption may lead to an observable state change in all processes, and the system may take a large amount of time to recover to a legitimate state. Our current work addresses this particular weakness by introducing the notion of *fault-containment* in the context of self-stabilization. In the context of digital systems, Nelson [32] defines fault-containment as the prevention of error propagation across defined boundaries. Our goal is to design *fault-containing, self-stabilizing* protocols, that is, protocols that, in addition to being self-stabilizing, have the ability to contain "small-scale" transient faults within some defined boundaries. By a "small-scale" fault we mean a transient fault that affects a small number of components physically close together in the system. This notion will be made more precise later in the paper. Suppose that α is a state of a system arising from a small-scale fault. Then, the property of fault-containment guarantees that during recovery from α: (a) the total number of observable state changes by all components in the system is small, and (b) only a small subset of components, physically close to the faulty components, change their local states. Thus the boundaries within which we wish to contain the effects of faults are spatial as well as temporal. Fault-containment guarantees that to any user of the system, small-scale faults are "almost" masked, and after such a fault occurs, the system is repaired quickly with minimal seepage of the faults. In addition to ensuring the containment of small-scale faults, it is desirable that the system also provide the broader guarantee of self-stabilization, since arbitrary large-scale faults may occur occasionally. A formal definition of fault-containment in the context of self-stabilizing systems is presented in Sect. 4.

Small-scale faults can be thought of as a class of faults distinct from the class of arbitrary transient faults and therefore fault-containing self-stabilizing systems can be thought of as examples of systems that provide *multitolerance*. Multitolerance refers to the ability of a system to tolerate multiple fault-classes, each in a possibly different way. Arora and Kulkarni [5] propose a component based method for designing multitolerance. However, in Arora and Kulkarni's method, the individual components must be non-interfering in a certain sense. In our context, this method might involve starting with a self-stabilization component and adding to it a fault-containment component. Unfortunately, the fault-containment component and the self-stabilization component typically interfere with each other adversely because the properties of self-stabilization and fault-containment seem to be inherently in conflict with each other. Hence, even though fault-containing self-stabilizing systems are examples of multitolerant systems, Arora and Kulkarni's method cannot be used to design such systems. The design of fault-containing self-stabilizing systems in an asynchronous network is made more difficult by the fact that there is little control on the order of moves by processes. For example, if the neighbor of a faulty process makes a move before the faulty process repairs its fault, then the fault may spread through the system before it is eventually repaired. The conflict between self-stabilization and fault-containment is the most fundamental problem faced during the addition of the property of fault-containment to self-stabilizing protocols. This problem is formally identified and solved in this paper.

The problem of containing the effects of small-scale transient faults is rapidly assuming importance for two reasons: (a) the dramatic growth in network sizes and (b) the fact that in practice, a transient fault usually corrupts a small number of components. For example, consider a broadcasting protocol that uses a spanning tree computed by an underlying self-stabilizing protocol (see [12] for an example of a self-stabilizing spanning tree protocol). A transient fault at a single process, say i, that corrupts the spanning tree information local to i may contaminate the spanning tree information in a large portion of the system, if the fault is not contained. The faulty spanning tree could significantly and adversely affect the operation of the broadcasting protocol across a large portion of the network. The damage to the broadcasting operation could be in terms of lost messages or a large number of unnecessary messages. So our goal is to tightly contain the effects of small-scale transient faults. What "tightly" exactly means depends on the context and the application. For example, suppose that α is a state obtained from a legitimate state by a transient fault at a single process. In one context, tight fault containment could mean that, starting in α, observable disruption of the system state persists for at most $O(1)$ time; in another context tight fault containment could mean that during recovery from α, only processes within $O(1)$ distance of the faulty process are allowed to make observable changes in their local state. In any case, tight fault-containment results in a self-stabilizing distributed system in which the disruption caused by small-scale faults is “almost” invisible and as a result adverse effects of such faults are minimal.

1.1 Contributions

Our first contribution lies in formalizing the notion of fault containment within the context of self-stabilization. As far as we know, this is the first attempt to investigate fault-containment systematically in the context of self-stabilization. We identify and formally define various metrics to evaluate the fault-containment and self-stabilization properties of a protocol, and based on these, give a formal definition of fault-containing self-stabilizing protocols. Informally, let a *k-faulty* state be an illegitimate state obtained from a legitimate state by transient faults that arbitrarily corrupt the local state of k processes. In this paper, we focus on fault-containment from 1-faulty states. However, our definitions extend to the case of k-faulty states as well.

Our second contribution is concerned with the methodology of designing fault-containing self-stabilizing protocols. The main question here is whether we can systematically construct from self-stabilizing protocols, equivalent protocols that are fault-containing, in addition to being self-stabilizing. This problem is important since self-stabilizing protocols already exist for a large number of problems. We formalize this problem as one of constructing a transformer that can transform any self-stabilizing protocol into an equivalent fault-containing self-stabilizing protocol. As mentioned earlier, this problem is difficult because of what seem to be inherently conflicting demands that the properties of self-stabilization and fault-containment place on a system. We solve the problem by presenting a transformer $\mathcal{T}$ that converts a non-reactive self-stabilizing protocol P into an equivalent fault-containing self-stabilizing protocol Q that repairs a 1-faulty state in $O(1)$ time, and with only the faulty process making an observable state change. For many instances of P, the output Q produced by the transformer has $O(1)$ space overhead per process. Thus, the single transient fault is contained as tightly as possible, temporally as well as spatially. Features of Q worth emphasizing are that it runs on an asynchronous network and only requires a small amount of extra space per process for fault-containment. To get around problems introduced by asynchrony and also as a way of distinguishing between single process faults and other more substantial faults, we use a novel timer-based technique in Q. This timer protocol is self-stabilizing and we believe is of independent interest in distributed computing.

1.2 Related work

Of late, there has been growing interest among researchers in constructing protocols that are not only self-stabilizing, but also provide certain guarantees during convergence from certain states. Gouda and Schneider [26] propose a stabilizing algorithm for constructing a maximum flow tree in a network; the algorithm allows arc capacities to change and ensures that even while a new maximum flow tree is being constructed in response to the new capacities, a flow tree is maintained. Dolev and Herman [17] construct “superstabilizing” protocols that, in addition to being self-stabilizing, guarantee that during convergence from states that arise from legitimate states by small-scale topology changes such as the crash or recovery of a processor or a link, certain *passage predicates* will be satisfied. Thus, superstabilizing protocols provide some guarantees during convergence, in contrast to self-stabilizing protocols that provide no guarantees on system behavior during convergence. However, Dolev and Herman assume that any such topology change is accompanied by a signal to neighboring processors. In the context of fault-containing self-stabilizing protocols, such signals or interrupts to distinguish between a small scale fault (from which we want fault-containment) and an arbitrary fault (from which we want only self-stabilization) may not be available. In fact, it is the need to distinguish between two types of faults that makes the design of fault-containing self-stabilizing protocols difficult. Hence, even though fault-containing self-stabilizing protocols are similar to superstabilizing protocols in the sense that they also provide additional guarantees during convergence from certain types of faults, the methods of [17] are not directly applicable to fault-containment. For several specific problems, self-stabilizing protocols that provide certain fault-containment properties from 1-faulty states have been presented: [19] solves the problem of electing a leader in a ring, [21] solves the problem of constructing a spanning tree, and [22] solves a the problem of constructing a breadth-first search spanning tree. Herman [27] presents a self-stabilizing protocol for mutual exclusion on a ring that contains the effect of any spurious tokens that may have been generated by a single-process fault. Kutten and Patt-Shamir [28] mention an asynchronous self-stabilizing algorithm for the persistent bit problem that recovers in time proportional to the number of faults. Beauquier, Genolini, and Kutten [9] present a token-based mutual exclusion protocol that can recover from k faults in $O(k)$ time.

Some recent papers address the bigger question of how to systematically construct fault-containing protocols from protocols that are not fault-containing. Kutten and Peleg [30] present a class of protocols for a synchronous network for which the recovery time is proportional to the number of transient faults. However, their protocols are not self-stabilizing. But the attempt by the authors to link recovery time to the severity of the fault is an important contribution. Kutten and Patt-Shamir [29] present a self-stabilizing algorithm for the persistent bit problem, where the goal is to retain the value of a common replicated bit across the system in spite of transient faults. Kutten and Patt-Shamir use this algorithm as the basis of a transformer that takes as input a non-reactive, possibly non-stabilizing protocol and produces as output an equivalent self-stabilizing protocol that recovers in time proportional to the number of faults from any state in which at most half the processes are corrupted by transient faults. Again, the resulting protocol works only on synchronous systems. An alternate method of stabilizing an asynchronous system in time commensurate with the extent of failure is proposed by Ghosh and He in [24]. It guarantees recovery from single faults in $O(1)$ time, but when the number of failures increases, the number of processes that makes observable state changes may grow exponentially. Thus the effect of the fault is not tightly contained within a small neighborhood around the fault. The recent work of Afek and Dolev [1] describe a transformer that takes as input a synchronous distributed protocol and produces as output an equivalent synchronous, self-stabilizing version that has the ability to locally repair faults in *expected* time proportional to the largest diameter of a faulty region. Afek and Dolev drop the requirement of Kutten and Patt-Shamir that at most half the processes be corrupted by a transient fault, and their transformer also handles interactive protocols. However, the fault-containment provided by Afek and Dolev's technique uses error-correcting codes and is probabilistic, and their protocols work only on synchronous systems. Running these synchronous protocols in an asynchronous environment is not merely a matter of using a synchronizer; the synchronizer itself has to be fault-containing in an asynchronous environment. The transformer we present in this paper is the first[1] attempt to automatically build fault-containing protocols in an asynchronous setting.

A crucial part of our transformer is a timer protocol. This is, in some superficial ways similar to the *synchronizer* protocols [6–8,13] that can be used to run synchronous protocols in an asynchronous environment. However, there are fundamental differences. The protocol in [6] is not self-stabilizing. The protocols in [7,8,13] are stabilizing. Of these, the protocol in [8] is very complicated. [7] presents two synchronizer protocols. The first one requires unbounded registers. The second protocol bypasses this problem by using a self-stabilizing reset mechanism whenever the register value reaches a predefined maximum. Similarly, [13] also provides two protocols for maintaining clock variables such that clocks of neighboring processes are synchronized. The first one requires an unbounded clock. The second one provides a bounded clock, where the bound must be greater than n^2 for an n-process system and is thus dependent on the network size. However, all the protocols guarantee only stabilization, and fail to meet the additional requirement of supporting fault-containment in the case of a single fault. In addition, all the protocols are non-terminating. The timer protocol we present is stabilizing, has special properties to achieve fault-containment in the case of a single fault in the system, and terminates when stabilization is completed.

1.3 Organization

The rest of the paper is organized as follows. Section 2 presents an example to motivate the issue of fault-containment in the context of self-stabilization; the example illustrates the difficulty of achieving fault-containment and self-stabilization in the same protocol. Section 3 presents our model of computation. Section 4 formally defines fault-containment in the context of self-stabilization. Section 5 considers the problem of automatic transformation of existing self-stabilizing protocols into fault-containing self-stabilizing protocols. A transformer is presented that takes as input a non-reactive self-stabilizing protocol, and automatically produces as output an equivalent fault-containing self-stabilizing protocol. Finally, Sect. 6 contains some concluding remarks.

2 A motivating example

To identify and emphasize the difficulties of combining self-stabilization with fault-containment, we present a simple example involving a transient fault that corrupts a single processor. The example is a simple self-stabilizing protocol [12] to construct a spanning tree of a network of n processes. A particular process r is designated as the root of the spanning tree to be constructed. The local state of each process i, $i \neq r$, is described by the tuple $\langle p, \ell \rangle$, where p is a variable that identifies i's parent in the tree and ℓ is a variable that denotes the distance in the tree between i and r. The process r has a single variable ℓ whose value is set to 0. Informally, each non-root process, asynchronously with any other process, perpetually checks its variables and adjusts them by applying one of two rules: (1) if $\ell < n$ and ℓ is not one greater than its parent's ℓ variable, then ℓ should be made one greater than that of its parent's ℓ variable; (2) if $\ell \geq n$ then p should be set to a new parent. Figure 1a shows a legitimate state for a small network (dashed lines are non-tree edges). The value of the variable ℓ for each process is shown beside the node and the parent variable p is shown by the arrow. At a legitimate state, only rule (1) applies and even when applied, it does not change the state of the process.

Now suppose that a single-process transient fault occurs in the legitimate state shown in Fig. 1a. This transient fault changes the distance variable ℓ at process x and the resulting state is shown in Fig. 1b. In this state, rule (1) is applicable at both x and y. If this rule executes first at x, the fault is

[1] A preliminary version of this paper appeared in the 15th Annual ACM Symposium on Principles of Distributed Computing, 1996.

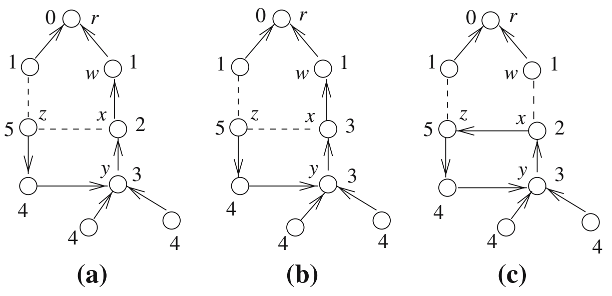

Fig. 1 An example showing the difficulty of containing a fault

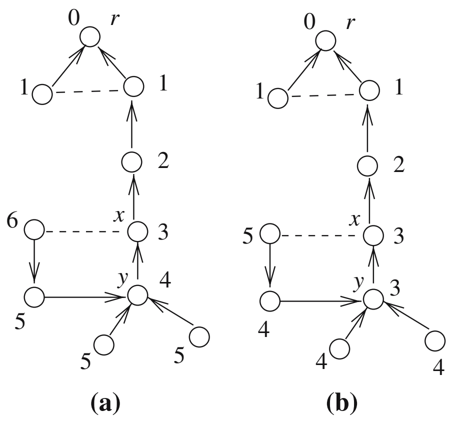

Fig. 2 An example to show the potential conflict between deadlock and fault-containment

instantly repaired and a legitimate state is reached; however if y executes first, then every descendant of y could adjust its distance variable ℓ as well, before the system returns to a legitimate state. This example hints that optimal fault containment might be achievable by proper scheduling of the actions of a protocol. However, achieving this scheduling is complicated by the fact that in a distributed system, processes have only local knowledge of the system. Consider the tree shown in Fig. 2a. Figure 2b shows a state resulting from faults in multiple processes that occur in the legitimate state shown in Fig. 2a. In particular, the faults change the distance variables of all processes in the subtree rooted at y, including the process y, as shown. Note that process y is the only process with an applicable rule that can change the local state of y, and hence the global state of the system. Hence, process y needs to change its state by an application of rule (1) in order to start the recovery to a legitimate state. However, the local knowledge of the system available to y is exactly the same as in the case of the tree shown in Fig. 1b. Thus, with the same local knowledge available to a process y, we require that in one case y moves to avoid deadlock, but require that in another case y does not move so as to achieve fault-containment. One implication of this is that there is a thin line between fault-containment and deadlock: on one hand, if a process chooses not to move, then the system might be deadlocked; on the other hand, if it chooses to move, then the system might not be fault-containing. A solution to this problem could be to extend the states of processes by adding auxiliary information so that each process knows more. But there is danger here as well, since a transient fault can corrupt the auxiliary information and lead to an equally unreliable local state.

Figure 1c shows another example of a state obtained by a single-process transient fault at the legitimate state shown in Fig. 1a. This fault causes process x to change its parent variable p from w to z and the result is a cycle. A legitimate state will be obtained after a number of rule executions by all processes in the cycle. Intuitively, one might hope that a transient fault at a single process can be repaired by actions only at that process, but this example shows stabilizing protocols may not have actions capable of such local repair. In particular, to recover from the 1-faulty global state shown in Fig. 1c, we would like x to reset its parent variable back to w immediately. However neither of the two rules in the protocol allow x to take such an action. To overcome this deficiency, one might attempt to add actions explicitly for local repair following a single-process transient fault. However this leads to other dangers. From local information, processes cannot distinguish between states in which it is better to execute the original protocol's actions versus states in which it is better to execute the newly added local repair actions. Executing local repair actions at the wrong time might undo some of the progress made by the original protocol and as a result the new protocol may not even be self-stabilizing.

The spanning tree example illustrates the potential conflict between self-stabilization and fault-containment — trying to achieve fault-containment may destroy the self-stabilization property of the system, and vice-versa. The essential difficulty in constructing self-stabilizing protocols that also achieve fault-containment is that based on local knowledge, processes cannot always unambiguously decide the extent of failure in the system. Hence, a process cannot always decide the proper action to take to achieve both fault-containment and self-stabilization.

We will next formalize the notion of fault-containment in the context of self-stabilization. As mentioned earlier in this paper, we will focus on fault-containment from single transient faults. We first describe our model of computation, and present definitions relative to this model. However, it is important to note that the scope of the definitions is not restricted to any one particular model of computation. The definitions can be easily extended to other models of computation, as long as the notion of a single fault is well-defined in that model. Moreover, the definitions can also be easily extended to the case when there is a need to contain the effects of not one, but k faults, for some fixed $k > 1$.

3 Model of computation

A distributed system is modeled by a network consisting of a set of nodes pairwise connected by a set of edges. Each node in the network corresponds to a process, and each edge between two nodes corresponds to a bidirectional communication link between the two nodes. Each process has a set of local variables. Let V be the set of all processes in the network and let $|V| = n$. A local variable belonging to a process i can be read by i and any of its neighbors, but can be written into by i only. Thus, a process can directly communicate with only its neighbors in the network by reading their local variables, and writing into its own local variables. The values of the local variables of a process constitute the *local state* of that process. The *global state* of the system is the collection of local states of all processes in the system.

Processes execute their protocols asynchronously. The protocol of each process i consists of a set of guarded statements of the form $G \rightarrow A$, where G, called the *guard*, is a boolean predicate involving the local variables of i and that of its neighbors, and A, called an *action*, is an assignment of values to one or more variables of i. The action is executed only if the corresponding guard is evaluated to be true. If multiple guards are true at a process, an action corresponding to an enabled guard is chosen non-deterministically from among the actions corresponding to all enabled guards. No fairness assumptions are made in choosing a guard from all enabled guards. The execution of a guarded statement (the evaluation of the guard and the execution of the action) is assumed to be serialized between any pair of processes that are neighbors. The execution of guarded statements at processes that are not neighbors may occur simultaneously. Serializing the actions of neighbors along with the fact that no fairness assumption is required allows us to think of the execution of the system as being completely serialized. This is of course equivalent to assuming a central daemon that serializes the execution of all guarded statements. So for simplifying our proofs we assume a central daemon, while noting that as mentioned above, in our model this can be implemented using local control. In this paper, we do not address how this local control can be implemented in a fault-contained manner. In the rest of this paper, we will refer to the atomic execution of a guarded statement by a process as a *move* by that process. An execution sequence of the system is a finite or infinite sequence of global states of the system that satisfies two properties: (a) if the sequence is finite, then in the last state of the sequence, no process has an enabled guard, and (b) if s and s' are two consecutive states in the sequence, then s and s' are distinct, and there exists a process i such that i has an enabled guard in s and execution of the corresponding action results in the state s' (the execution sequence, in this case, is said to contain a move by process i). The running time of a protocol is measured in *rounds*. Given any contiguous subsequence X of an execution sequence, X can be uniquely partitioned into rounds as follows. Let $X_0 = X$. Then for $i = 1, 2, 3, \ldots$, round i, denoted by r_i, is the minimum prefix of X_{i-1} such that for every process j, either r_i contains a state in which all guards at process j are disabled, or r_i contains a move by j. For each $i = 1, 2, 3, \ldots$, the sequence X_i is such that $X_{i-1} = r_i X_i$. Thus X_i is obtained from X_{i-1} by deleting the prefix r_i. The intuition is that each round of an execution sequence is the minimal prefix in which all processes get a chance to execute. However, some processes squander their chance to execute because when they choose to be scheduled, none of their guards are enabled.

4 Fault-containment and self-stabilization

To present a formal definition of a fault-containing self-stabilizing protocol, we adopt the following approach. We first consider the property of self-stabilization and the property of fault-containment of a protocol separately. We then identify and define various metrics to evaluate these two properties. Finally, we define fault-containing self-stabilizing protocols as those protocols that satisfy certain upper bounds on these metrics.

4.1 Measures of Self-stabilization

Let P be a protocol executed by a distributed system and let L be a predicate defined over the set of global states of the system. The protocol P is said to *converge* to L if every execution sequence of P (independent of the initial state) contains a global state satisfying L. L is said to be *closed* in P if every execution sequence of P that starts in a state satisfying L only contains states satisfying L. P is said to be *self-stabilizing* if P converges to L and L is closed in P. In particular, we say that P *stabilizes to* L. The set of states specified by L is called the set of *legitimate states* of P. A state that is not legitimate is called an *illegitimate state*. Note that any protocol P trivially stabilizes to the predicate $\texttt{TRUE}$, which corresponds to the set of all global states of the system. For any self-stabilizing protocol P we denote by $\texttt{States}(P)$ the set of global states of the system and we denote by $\texttt{Legit}(P)$ the subset of $\texttt{States}(P)$ that contains exactly the legitimate states of P. That is, if P stabilizes to L, then $\texttt{Legit}(P) = L$.

Partition the local state of a process into two parts: a *primary* part and a *secondary* part. The local state of a process i can therefore be denoted by the ordered pair $\langle p_i, s_i \rangle$, where p_i and s_i denote the primary part and the secondary part respectively of the local state of i. Correspondingly, each global state of the system can be written as an ordered pair $\langle p, s \rangle$, where p (respectively, s) is the collection of primary (respectively, secondary) parts of the local states of all processes in the system. We will refer to p as a *primary state* of P and to s as a *secondary state* of P. We will use $\texttt{States}_p(P)$ to denote the set of primary parts of all states in $\texttt{States}(P)$. The motivation for partitioning the state of each process, in this manner into a primary and a secondary part, is the following. Typically, a protocol produces an output that is used either by some other system or directly by a human user. Usually, the output of a protocol is represented by a few variables, while the rest of the variables simply assist in computing the output. We use p_i, the primary part of the state of process i, to denote the output of process i; s_i, the secondary part, denotes the rest of the state of process i. Correspondingly, p denotes the output of P and s denotes the rest of the global state of P. The exact definition of what constitutes the primary part and what constitutes the secondary part of a state depends on the particular application at hand. In fact, the secondary state s could be empty. For example, [7] presents a self-stabilizing protocol whose primary state stabilizes in $O(D)$ time, where D is the diameter of the network, but whose secondary state takes much longer to stabilize.

Assuming that each global state of the system is partitioned into primary and secondary states, the notion of legitimacy can be extended to the set of primary states, $\texttt{States}_p(P)$, as well. That is, $\texttt{States}_p(P)$ can be partitioned into a subset of legitimate states and a subset of illegitimate states so that independent of the secondary state of P, users of P are satisfied once the primary state of P becomes legitimate. This notion can be formalized as follows. Let $\texttt{Legit}_p(P)$ denote a set of primary states of P such that:

(a) For any global state $\langle p, s\rangle$, $\langle p, s\rangle \in \texttt{Legit}(P) \Rightarrow p \in \texttt{Legit}_p(P)$.
(b) $\texttt{Legit}_p(P)$ is closed in P (that is, every execution sequence of the system that starts in a state whose primary state is in $\texttt{Legit}_p(P)$, contains only states whose primary parts are in $\texttt{Legit}_p(P)$).

Of course, the predicate $\texttt{Legit}_p(P)$ should be representative of exactly those values of the output that the user considers correct. This, in turn, depends on the application at hand. Thus, our view is that starting from an arbitrary state, P first reaches a state whose primary part is in $\texttt{Legit}_p(P)$ and then eventually reaches a state in $\texttt{Legit}(P)$. Thus, P may pass through states $\langle p, s\rangle$ for which $p \in \texttt{Legit}_p(P)$ and $\langle p, s\rangle \notin \texttt{Legit}(P)$. The user of P is satisfied with such states, but such states are not yet legitimate – the secondary state has to undergo more changes before the entire state becomes legitimate.

We now define the following metrics as a measure of the stabilization property of a system.

Definition 1 (*Stabilization time*:) The worst case time, starting from any initial state, to reach a state in $\texttt{Legit}(P)$. The stabilization time of a protocol P is denoted $T(P)$.

Definition 2 (*Primary stabilization time*:) The worst case time, starting from any initial state, to reach a state whose primary part is in $\texttt{Legit}_p(P)$. The primary stabilization time of a protocol P is denoted $T_p(P)$.

Definition 3 (*Stabilization space*:) The maximum space used by any process in a legitimate state. The stabilization space of a protocol P is denoted $S(P)$.

Our measurement of space at a legitimate state (as opposed to space used during convergence to a legitimate state) is motivated by two previous papers on self-stabilization [2,33].

Most self-stabilizing systems do not distinguish between the primary and the secondary parts of global states, and are only concerned with the entire state becoming legitimate quickly. In fact, we know of only one example of a self-stabilizing system in which a distinction between stabilization of the primary state and stabilization of the entire state has been made [7]. As we will demonstrate, in the context of fault-containment, the distinction between primary and secondary states of a system is extremely important. The fault-containing self-stabilizing protocols that we construct in the sequel have the ability to start from a 1-faulty state and repair the primary state quickly. However, the secondary state, containing variables that help in detecting and repairing single process faults, takes much longer to become legitimate. In fact, it is an open question whether the stabilization time of our protocol (as opposed to the primary stabilization time) starting from a 1-faulty state can be improved without larger space overhead. In the next section, we identify and define metrics for evaluating the fault-containment property of a protocol.

4.2 Measures of fault-containment

Let a 1-*faulty state* of the system be defined as an illegitimate state that differs from some legitimate state in the local state of exactly one process in the system. This definition is similar to the notion of Hamming distance used in error-correcting codes and used in other papers such as [17,29]. Thus a 1-faulty state could have been caused by the corruption of the local state of exactly one process, though the reader should note that a 1-faulty state could also be the result of faults at more than one process. Our goal is to repair a 1-faulty state by changing the primary part of the local state of exactly one process.

We propose the following metrics to evaluate the property of fault-containment of a protocol.

Definition 4 (*Containment time*:) The worst case time, starting from any 1-faulty state, for a protocol to reach a state whose primary part is in $\texttt{Legit}_p(P)$. The containment time of a protocol P is denoted $T_p^1(P)$.

Definition 5 (*Contamination number*:) The worst case number of processes that change the primary part of their local state during recovery from any 1-faulty state to a legitimate state. The contamination number of a protocol P is denoted $N^1(P)$.

Definition 6 (*Fault gap*:) The worst case time, starting from any 1-faulty state, to reach state in $\texttt{Legit}(P)$. The fault gap of a protocol P is denoted $T^1(P)$.

Definition 7 (*Containment space*:) The maximum size of the secondary part of the local state of any process at a legitimate state. The containment space of a protocol P is denoted $S^1(P)$.

Thus, the containment time is a measure of how long it takes for a 1-faulty state to be repaired as far as the primary state of the system is concerned. This is analogous to the primary stabilization time of P if it is restricted to start from 1-faulty states. Note that containment time is defined with respect to a suitably defined predicate of interest, $\texttt{Legit}_p(P)$, and therefore $T_p^1(P)$ may change if $\texttt{Legit}_p(P)$ is changed. Since only the primary part of the state is observed and used, contamination number is a measure of how many processes make observable changes to their local states, before the 1-faulty state is repaired. Ideally, we would like only the faulty process to change the primary portion of its state before L_p is established, and in such a case the contamination number would be one.

The first two metrics defined above consider efficient recovery of the primary part of the protocol only. However, as noted earlier, the predicate $\texttt{Legit}_p(P)$ is established does not mean that the system is in a legitimate state. That is, the system may be in a state $\langle p, s\rangle$ where $p \in \texttt{Legit}_p(P)$ and $\langle p, s\rangle \notin \texttt{Legit}(P)$. In such a state, if a single-process fault occurs then, the system provides no guarantee (beyond worst case stabilization time) as to how quickly this fault will get repaired. This is because even though a state $\langle p, s\rangle$ for which $p \in \texttt{Legit}_p(P)$ and $\langle p, s\rangle \notin \texttt{Legit}(P)$, might seem like a legitimate state to a user, in reality it may be nowhere "close" to being legitimate and a single process fault in such a state may not yield a 1-faulty state. Therefore, for the system to repair a single-process fault quickly, that fault may have to be separated by at least $T^1(P)$ time from the most recent fault. This motivates the name "fault-gap" for $T^1(P)$. Note that the fault gap is at least as large as the containment time and in general can be much larger.

Finally, the metric, containment space, measures the space overhead incurred in achieving fault-containment. Note that similar to the definition of stabilization space, we measure space used at a legitimate state, as opposed to space used during convergence to a legitimate state. While this by itself does not specify any bounds on the space requirement during convergence, we will show that our implementations use only a bounded amount of space.

The reader should note that the definitions provided above can be extended in an obvious manner to the case of k-faulty states.

4.3 Definition of fault-containing self-stabilization

Based on the metrics defined in the above section, we now give the following definition of a fault-containing self-stabilizing protocol. The definition requires that the protocol be self-stabilizing and in addition satisfy certain upper bounds on some of the metrics defined for fault-containment.

Definition 8 (*Fault-containing self-stabilizing protocol*:) A protocol P that is self-stabilizing and in addition satisfies $T_p^1(P) = O(1)$ and $N^1(P) = O(1)$.

According to the definition above, for a fault-containing self-stabilizing protocol, the user will observe the disruption caused by a single process fault for only a constant amount of time (independent of the size of the network or the size of the neighborhood of the faulty process). Furthermore, the user will only observe changes in the state of a constant number of processes. Thus the effect of a single process fault is almost invisible to a user.

Note that depending on the aims of a particular application, alternate definitions of a fault-containing self-stabilizing protocol are possible. For example, if fast convergence of the system to a legitimate state is of more importance than just masking the effect of the fault from users, then bounds on the fault-gap can be used to define a fault-containing self-stabilizing protocol. Ideally, it is desirable to have a constant upper bound on all four of containment time, contamination number, fault-gap, and containment space. However, in general, it is hard to design such a protocol for an arbitrary network.

5 General transformation techniques

As mentioned earlier our goal is not to construct fault-containing self-stabilizing protocols from scratch. Our goal is to start with an existing self-stabilizing protocol for a problem, and then modify it by adding the property of fault-containment. The modification of an existing self-stabilizing protocol simplifies the task of designing and verifying the self-stabilization property of the derived protocol. This view is confirmed by our design of three self-stabilizing protocols [19,21,22] that provide certain fault-containment properties from 1-faulty states; all three protocols are constructed from existing self-stabilizing protocols. However, in all three cases, modifying the existing protocol and proving the correctness of the resulting protocol required the use of problem-specific information. Since self-stabilizing protocols have been designed for a large number of problems, an extremely important problem in this context is designing a transformer that takes as input a self-stabilizing protocol, and automatically produces as output an equivalent fault-containing self-stabilizing protocol. Such a transformer necessarily cannot use any problem-specific information. In this section, we consider the problem of constructing a transformer. We first formally define the transformation problem. We then present a transformer for the class of non-reactive self-stabilizing protocols. Note that in a non-reactive protocol [31], no guard is enabled in the final state. Hence, the existence of an enabled guard indicates an illegitimate global state.

5.1 The transformation problem

In order to state the transformation problem, we formalize the notion of equivalence between a pair of self-stabilizing protocols in the context of fault-containment.

Definition 9 Let P be a self-stabilizing protocol. A fault-containing self-stabilizing protocol Q is *equivalent* to P if

(a) $\mathtt{States}_p(Q) = \mathtt{States}(P)$.
(b) $\mathtt{Legit}_p(Q) = \mathtt{Legit}(P)$.

Intuitively, if Q is a fault-containing self-stabilizing protocol equivalent to P, then Q behaves like P except that Q is fault-containing. A user can barely notice the disruption in Q caused by a single process fault. On the other hand, P may not be fault-containing and it is possible that a single process fault causes substantial disruption in P. Note that we are being conservative in assigning the entire state of P as the primary state of Q. It is possible that the primary state of P, namely that part of the state that the user is really interested in, is a small part of the entire state of the system. In this case, $T_p^1(Q)$ is an upper bound on the worst case time that Q takes to go from a 1-faulty state to a state in which the user considers the output correct. The secondary state of Q may contain auxiliary variables used for fault-containment. Once the primary state of Q has become legitimate (that is, the primary state belongs to $\mathtt{Legit}_p(Q)$) then the secondary state will not affect it and eventually the secondary state will also become legitimate. The *Transformation Problem* can now be defined as follows.

Definition 10 Construct a transformer $\mathcal{T}$ that takes as input a self-stabilizing protocol P and produces as output a fault-containing self-stabilizing protocol Q equivalent to P.

Independent of whether P is fault-containing or not, the transformer $\mathcal{T}$ constructs a protocol Q that essentially does everything that P does and in addition, guarantees fault-containment. Construction of $\mathcal{T}$ is the topic of the rest of the paper.

5.2 A solution to the transformation problem

In this section we present a solution to the transformation problem. We construct a transformer $\mathcal{T}$ that takes a non-reactive self-stabilizing protocol P and maps it into an equivalent non-reactive fault-containing self-stabilizing protocol Q.

To simplify presentation, we assume that in protocol P, each process i executes a single guarded statement of the form $G_i \rightarrow A_i$. Multiple guarded statements $G^1 \rightarrow A^1, G^2 \rightarrow A^2, \ldots$ at a process can always be written as a single guarded statement with guard $G^1 \vee G^2 \vee \cdots$ and action *if* G^1 *then* A^1 *else if* G^2 *then* $A^2 \ldots$ The exact definitions of G_i and A_i depend on the particular protocol P being considered. The main idea in constructing Q from P is the following. View the protocol Q as consisting of two protocols C and C' cascaded together, such that C executes first, followed by C'. For now, suppose that C is a synchronous non-self-stabilizing protocol whose behavior is as follows: if started in a 1-faulty state of Q, C takes the system into a state satisfying $\mathtt{Legit}_p(Q) \equiv \mathtt{Legit}(P)$ in $O(1)$ synchronous steps, with only $O(1)$ processes changing the primary portion of the state. C' can be thought of as the protocol P along with some additional "book-keeping" code necessary to establish the truth of $\mathtt{Legit}(Q)$ after $\mathtt{Legit}_p(Q)$ is true. From a 1-faulty state, C executes first and in $O(1)$ time repairs the fault and takes the system into a state satisfying $\mathtt{Legit}_p(Q)$. Then C' executes and takes the system into a state satisfying $\mathtt{Legit}(Q)$. Note that the truth of $\mathtt{Legit}_p(Q)$ implies the truth of $\mathtt{Legit}(P)$, and since P is non-reactive, the truth of $\mathtt{Legit}(P)$ implies that no guards of P are enabled. Hence, in the execution of C', no process can change the primary portion of the state and $\mathtt{Legit}_p(Q)$ cannot be made false again. Thus, the containment time and the contamination number of Q are both $O(1)$. If Q is started in a state that is not 1-faulty, then first C executes and does nothing useful. Subsequently, C' executes and takes the system into a state satisfying $\mathtt{Legit}(P)$. Hence, Q is a fault-containing self-stabilizing protocol.

Since protocol Q executes in an asynchronous network, the implementation of the above idea requires mechanisms that provide two levels of synchronization. Firstly, since C is a synchronous protocol, it is necessary to provide synchronization between different steps of C. Secondly, it is necessary to synchronize the protocols C and C' such that a process starts executing C' only after all processes have finished executing the last step of C. However, since a state change by a process depends only on its own local state and the local state of its neighbors, it suffices to ensure synchronization between neighboring processes only. In other words, for any process i, it suffices to ensure that process i does not execute a step of C until i and all neighbors of i have finished the execution of the previous step of C, and process i does not start executing C' until i and all its neighbors have finished executing the last step of C. This synchronization is provided by a *timer* protocol described in the following subsection.

As we mentioned earlier, this timer protocol is in some ways similar to the *synchronizer* protocols in [6–8]. However, all the protocols above guarantee only stabilization, and fail to meet the additional requirement of supporting fault-containment in the case of a single fault. Specifically, in the case of a single fault, the synchronizer protocols do not guarantee that C' runs after C. In fact, we can construct fault scenarios and execution sequences for these protocols in which C' is executed before C and fail to achieve fault-containment. The timer protocol we present is stabilizing, has special properties to achieve fault-containment in the case of a single fault in the system, and terminates when stabilization is completed. We describe the timer protocol in the next section.

5.2.1 Timer protocol

The timer protocol maintains a timer variable t_i at each process i. This variable can take any integer value in the range $[0..M]$, where M is a sufficiently large positive integer. We will discuss what the value of M should be later in this paper. The timer variable t_i is *consistent* if its value differs by at most one from the value of the timer variable of each of its neighbors; otherwise t_i is said to be *inconsistent*. The goal of the timer protocol is to make all timer variables in the system consistent while they have sufficiently large values, and then decrement them without losing consistency so that eventually all timer variables have the value 0. The timer values decrementing consistently essentially simulates a global clock that provides the necessary mechanism for synchronizing neighboring processes.

The implementation of the timer is shown in Figure 3. The protocol consists of two guarded statements, one that sets the timer variable of a process to M and the other that decrements the timer variable by one. The timer variable of a process i, t_i, is set to M if either of the following two predicates is true:

Condition 1 $\equiv (t_i \neq M) \wedge (\exists j \in N_i : (t_i - t_j > 1) \wedge (t_j < M - n))$
Condition 2 $\equiv (t_i < M - n) \wedge (\exists j \in N_i : t_j = M)$

Thus the predicate *raise*(t_i), that appears in statement S_1 in Fig. 3 is the disjunction of the above two predicates. Note that $raise(t_i)$ is true only when t_i is inconsistent because the truth of either of Condition 1 or Condition 2 implies the inconsistency of t_i.

```
        do
[S1]        raise(t_i)       →  t_i := M;
[S2]    □   decrement(t_i)   →  t_i := t_i − 1;
        od
```

Fig. 3 The implementation of timer. Program for process i

The timer variable of a process is decremented if either of the following predicates is true:

Condition 3 $\equiv (t_i > 0) \wedge (\forall j \in N_i : 0 \leq t_i - t_j \leq 1)$
Condition 4 $\equiv (\forall j \in N_i : t_i \geq t_j \wedge (t_j \geq M - n))$

Thus the predicate *decrement*(t_i), shown in statement S_2 in Fig. 3, is the disjunction of the above two predicates. Note that both conditions require t_i to be no less than the timer value of any neighbor. Condition 3 requires that t_i be consistent, while Condition 4 allows t_i to be decremented even when it is inconsistent, provided that the neighbors of i have timer values in the range $[M-n..M]$ and t_i is the largest timer value in the neighborhood. All guards in the timer protocol become false when all timer values are 0. The important feature of the timer protocol is that it provides synchronization between neighboring processes. In particular, the only way a timer value can come down to 0 is through decrement actions that are "synchronized" with neighbors over the interval $M - n$ down to 0. Choosing M appropriately allows us to make this interval sufficiently large.

5.2.2 The protocol Q

The variables of the protocol Q are the variables of the protocol P (the self-stabilizing protocol input to the transformer), the timer variables, along with any additional variables used by protocol C. In a legitimate state of Q, characterized by the predicate $\texttt{Legit}(Q)$, the variables of P are in a legitimate state characterized by the predicate $\texttt{Legit}_p(Q) \equiv \texttt{Legit}(P)$ (that is, G_i is false for every i), the timer variables are all equal to 0, and any additional variables of C are in a legitimate state. To make this more precise, partition the variables that the protocol C uses into two sets: the variables of P and the timer variables belong to one set and the rest of the variables belong to the other set, which we shall call the *book-keeping* variables of C. Let the predicate $\texttt{Legit}_b(C)$ characterize global states in which the book-keeping variables of C are legitimate. Thus, $\texttt{Legit}(Q) \equiv \texttt{Legit}_p(Q) \wedge \texttt{Legit}_b(C) \wedge (\forall i \in V : t_i = 0)$. Assume that the predicate $\texttt{Legit}_b(C)$ can be written as $\texttt{Legit}_b(C) \equiv \forall i : \texttt{Legit}_b(C, i)$, where $\texttt{Legit}_b(C, i)$ is a predicate which is true if and only if the book-keeping variables of C at process i have values that are the same as their values in some legitimate state of Q. We will give two alternate implementations of the protocol C, and the corresponding definitions of $\texttt{Legit}_b(C)$ and $\texttt{Legit}_b(C, i)$, later in the paper (Sect. 5.4). Note that in a state satisfying $\texttt{Legit}_b(C)$, what the value of the book-keeping variables of C should be, may depend on the values of the variables of P.

Let the predicate *PorC_inconsistent*(i) be defined as
$PorC\_inconsistent(i) \equiv$
$$(t_i = 0 \ \wedge \ (\forall j \in N_i : t_j = 0) \ \wedge \ (G_i \ \vee \ \neg\texttt{Legit}_b(C, i))).$$
Thus, *PorC_inconsistent*(i) is true if the timer values of i and its neighbors are all zero, but either the variables of P or the book-keeping variables of C at i are different from what their values should be in a legitimate state of Q. The protocol Q is shown in Fig. 4. Note that the only change to the timer protocol is the weakening of the first guard by adding the predicate *PorC_inconsistent*(i), and the addition of an action to the second guarded statement. We now elaborate on each of the two guarded statements in the protocol.

- **Guarded statement** S_1**:** Suppose that a single-process fault occurs at a process i in a legitimate state of Q. If the fault corrupts t_i and sets it to a value greater than 1, then *raise*(t_i) is true. If the fault either does not corrupt t_i or sets t_i to 1, and corrupts either the variables of P or the book-keeping variables of C and changes any of them from their values in a legitimate state, then *PorC_inconsistent*(i) or $PorC\_inconsistent(j)$ for some $j \in N_i$ will be true eventually. In either case, the fault triggers process i or one of its neighbors to set its timer variable to M by an execution of guarded statement S_1. If a process $k \in \{i\} \cup N_i$ sets its timer to M, then $raise(t_\ell)$ becomes true for all $\ell \in N_k$ with $t_\ell = 0$. Any such process ℓ then executes S_1 and sets t_ℓ to M. In this manner, the initial fault at i acts as a signal that spreads across the entire network and all processes eventually set their timer variables to M. In case of arbitrary faults, multiple processes may set their timer variables to M independently. However, it will be shown that the timer variables of all processes become consistent with a sufficiently large value within a finite time. It is important for the correctness of protocol Q that the timer variables of all processes become consistent and then decrement down to 0 maintaining consistency over a sufficiently large interval of values. Note that consistent decrementing of the timer variables amounts to executing the actions of the protocol synchronously.
- **Guarded statement** S_2**:** When a process is ready to decrement its timer, it does so, but before that, depending on the value of its timer variable, the process executes an action. This is implemented by the second guarded statement in Fig. 4. We now describe *action*(t_i), which is the action performed by process i when it is ready to decrement its timer variable. Corresponding to different

```
       do
[S1]          raise(ti) ∨ PorC inconsistent(i)
                                  → ti := M;
[S2]       □  decrement(ti)     → action(ti); ti := ti − 1;
       od
```

Fig. 4 Protocol Q based on timer. Program for process i

values of t_i, different actions are performed and these are described below. Choosing *action*(t_i) appropriately for different values of t_i allows us to coordinate C and C', as desired. The timer variable range, $[1..M]$ is partitioned into the following subranges:

1. $[M - c + 1..M]$: Here c is the running time of C. If t_i is in this range, then i executes an appropriate action of the protocol C. In particular, if the c steps of the protocol C are consecutively numbered from 0 to $(c - 1)$, then i executes step 0 when $t_i = M$, step 1 when $t_i = M - 1$, step 2 when $t_i = M - 2$ and so on. After each execution of a step, t_i is decremented. Thus, when $t_i = M - c + 1$, the last step of C is executed, and t_i is decremented to $M - c$. It will be shown that starting from a 1-faulty state, t_i is decremented only when all neighbors of i have timer values equal to t_i or one less than t_i. Thus, by paying attention to the values of t_i while executing C, we have essentially synchronized C, even though the network is asynchronous. This is because a process i executes a step of C only when all neighbors have completed executing actions from the previous synchronous step.
2. $[b+1..M-a]$, for some $b \geq 1$, where $a = \max(c, n)$: If t_i is in this range, then i executes the action A_i provided G_i is true. Thus each process executes protocol P when its timer variable is in the range $[b+1..M-a]$. Note that for a particular n, if $c \geq n$, then this subrange is contiguous to the subrange $[M-c+1..M]$ above. However, if $c < n$, then no action is taken if t_i is in the intermediate subrange $[M - n + 1..M - c]$. For the first of the two implementations of C that we present later, the value of c is 3 and for the second implementation of C, the value of c is 11. The range $[b + 1..M - a]$ should be large enough to allow protocol P to reach legitimacy (that is, for $\texttt{Legit}_p(Q) \equiv \texttt{Legit}(P)$ to be established) after executing in the range. The value of b is determined by the book-keeping operations necessary to establish $\texttt{Legit}_b(C)$ at the end of the execution of P.
3. $[2..b]$: If t_i is in this range, then some book-keeping operations are performed. These operations restore the additional variables of C to legitimacy. Since in general, the values of any additional variables in C may depend on the values of the variables in P, these operations are performed after the variables in

P have achieved legitimacy and hence, do not change anymore. Whether these book-keeping operations are necessary or not depends on the implementation of the protocol C. The value of b is also dependent on the implementation of C and the necessary book-keeping operations. For example, in the first of the two implementations of C that we present later, b is 2. That is, book-keeping operations are performed by any process in the single step when its timer decrements from 2 to 1. The second implementation of C does not need any book-keeping operations.

4. $[1..1]$: If t_i is in this range (that is, when t_i is decremented from 1 to 0), then no action is taken.

The actions executed in the different subranges of the timer are shown in Fig. 5. The only restriction on the value of M is that it should be large enough to allow the protocol P to reach legitimacy when it executes in the range $[b+1..M-a]$, where $a = \max(c, n)$. Note that this implies that n is a lower bound on M. In case of a single fault, C executes first in the range $[M-c+1..M]$ to establish $\texttt{Legit}_p(Q)$ in constant time, with only a constant number of processes changing the primary portion of the state. Since the truth of $\texttt{Legit}_p(Q)$ implies that G_i is false for all i, no state change occurs when the timer is in the subrange $[b+1..M-c]$. Book-keeping operations are then performed in the subrange $[2..b]$, to set the book-keeping variables of C correctly, if necessary. These operations ensure the truth of $\texttt{Legit}_b(C)$. In case of an arbitrary fault, the execution of C may not be useful in any way. Subsequently, the execution of P in the subrange $[b+1..M-a]$ establishes $\texttt{Legit}_p(Q)$. The book-keeping operations are then performed to restore the truth of $\texttt{Legit}_b(C)$, if necessary, in the subrange $[2..b]$.

The per-process space overhead incurred by protocol Q is the sum of any space overhead incurred by protocol C and the space required to store the timer variable. Since the maximum value of the timer is M, the space required to store the timer is $O(\log M)$ bits. However, in a legitimate state, this can be further reduced by encoding the timer variable in a single bit. In such an implementation, a corruption of the timer variables can result in one or more timers having value 1, which will be immediately corrected by the corrupted timers decrementing their values to 0 (by guarded statement S_2). If any variables of P or C are corrupted as well, the predicate *PorC_inconsistent*(i) will be true for some i. In this case, space can be allocated dynamically for t_i to hold the value M. Similarly, space can be allocated dynamically for any other timer as well when they are incremented to M from 0. Later in the paper, we present two implementations of C. The first implementation works for all protocols P and it requires extra space at each process to store copies of local states of all neighboring processes. The second implementation works for a large class of protocols P, but requires $O(1)$ space overhead per process. In either case, the space overhead for C is small, and thus, the space overhead of Q in a legitimate state can be very small.

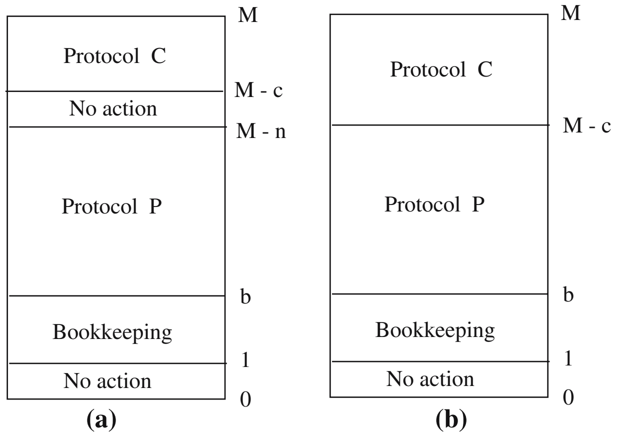

Fig. 5 Actions performed in various subranges of the timer when (i) $c < n$ and (ii) $c \geq n$

5.3 Proof of correctness

The proof of correctness of Q is divided into two parts, a proof of self-stabilization, and a proof of fault-containment. In this paper, we give an overview of the proof and state without proof, the main results. The complete proof of correctness can be found in [20]. We start with the following assumption.

Range Assumption: The value of M is large enough such that when any one process decrements its timer from M to 0, G_i is false and $\texttt{Legit}_b(C, i)$ is true for *every* process i in the system.

Note that by the definition of $decrement(t_i)$, the decrements of t_i from $M-n$ down to 0 must be synchronized with the timer values of the neighbors of i. Also, for technical convenience, we assume that $M-a$ is greater than 1, that is, each of the two ranges $[M-c+1..M]$ and $[b+1..M-a]$ are non-empty.

We show two implementations of the protocol C at the end of the paper that satisfies the following property:

[Fault-Containment Property] C is a synchronous protocol such that when C is started in a 1-faulty state with fault at process i, $\texttt{Legit}_p(Q)$ is established within c synchronous steps for some constant c with only the faulty process i changing the primary portion of the state. Moreover, only process i and its neighbors change state (either primary or secondary) before $\texttt{Legit}_p(Q)$ is established.

We will use this property of the protocol C in the proof of fault-containment of Q.

With respect to the protocol Q shown in Fig. 4, we will refer to the guard of statement S_1 in process i as $G_1(i)$ and the guard of statement S_2 in process i as $G_2(i)$ in the rest of this paper.

5.3.1 Proof of self-stabilization

We first define a *legitimate state* of Q.

Definition 11 A legitimate state of Q is a state in which for every process i, G_i is false, $\texttt{Legit}_b(C, i)$ is true, and $t_i = 0$.

Clearly, in such a state, no guard is enabled in the system. To prove the partial correctness of Q, we will show that if no guard is enabled in the system, then the system is in a legitimate state. We first show the following properties of the timer.

We first define a relaxed version of a consistent timer that we will call a *pseudo-consistent* timer.

Definition 12 The timer t_i at a process i is pseudo-consistent if either t_i is consistent, or if $t_i \geq M-n$ and for every $j \in N_i$, $t_j \geq M-n$.

Note that t_i may be pseudo-consistent even though its value differs by more than one from the timer t_j of some neighbor j of i, as long as for every $k \in \{i\} \cup N_i$, $M - n \leq t_k \leq M$. It is easy to verify that the predicate $decrement(t_i)$ is true if and only if $t_i \neq 0$, t_i is pseudo-consistent and $t_i \geq t_j$ for all $j \in N_i$. The system is in a *timer-consistent* state if the timer of every process in the system is pseudo-consistent. The following results can be easily verified.

Lemma 1 *If the system is in a timer-consistent state and $t_i \neq 0$ for some process i, then $decrement(t_j)$ is true for some process j.*

Lemma 2 *If the system is not in a timer-consistent state, then $raise(t_i)$ is true for some process i.*

The partial correctness of Q can be easily established from Lemmas 1 and 2.

Theorem 1 *The protocol Q is partially correct.*

We next prove the termination of Q. A move changing the timer value of a process can be divided into two classes: an *increasing* timer move is a move in which the timer is set to M, and a *decreasing* timer move is a move in which the timer is decremented by 1. Thus, in protocol Q, an increasing timer move is an execution of S_1, and a decreasing timer move is an execution of S_2. We will first show that the number of increasing timer moves is finite. This implies that the total number of moves is finite, since all remaining moves are decreasing timer moves, and the timer of a process cannot be decreased below 0.

An increasing move on a timer t_i can happen because of two reasons, either if *PorC_inconsistent*(i) is true or if *raise*(t_i) is true. If *PorC_inconsistent*(i) is true, we will call the corresponding increasing timer move a `Type 1` increasing move. All other increasing timer moves (that is, when *PorC_inconsistent*(i) is false but *raise*(t_i) is true) will be called `Type 2` increasing moves. Note that by definition of *raise*(t_i), t_i must be inconsistent for process i to execute a `Type 2` increasing move.

By definition, a `Type 2` increasing move by a process i occurs when *raise*(t_i) is true. Note that *raise*(t_i) is a disjunction of two predicates, $Y_i \equiv (t_i \neq M \wedge (\exists j \in N_i : (t_i - t_j > 1) \wedge (t_j < M - n)))$ and $Z_i \equiv (\exists j \in N_i : t_j = M \wedge (t_i < M - n))$. We will further subdivide `Type 2` increasing moves into two classes. A `Type 2(a)` increasing move is an execution of a `Type 2` increasing move when the predicate Y_i is true. All other `Type 2` increasing moves (that is, when Y_i is false but Z_i is true) are called `Type 2(b)` increasing moves.

The following lemmas bound the number of `Type 1` and `Type 2(a)` increasing moves. Lemma 3 follows from the fact that in order for a process i to execute two `Type 1` increasing moves, i has to decrease its timer from M (set after the first `Type 1` increasing move) down to 0 (in order for the second `Type 1` increasing move to occur). But then, by the **range assumption** on the value of M, G_j is false and $\texttt{Legit}_b(C, j)$ is true for all j after t_i is set to 0 and the second `Type 1` increasing move cannot occur.

Lemma 3 *A process can execute at most one* `Type 1` *increasing move.*

Lemma 4 *In any execution sequence, a process can execute at most one* `Type 2(a)` *increasing move.*

Thus, there is a finite prefix of the execution sequence X that contains all `Type 1` and `Type 2(a)` moves. Consider the suffix of X after the last execution of a `Type 1` or `Type 2(a)` increasing move by any process, and call it X'. Thus, X' contains only executions of `Type 2(b)` increasing moves, and decreasing moves. Unlike `Type 1` and `Type 2(a)` increasing moves, a process can execute more than one `Type 2(b)` increasing moves. However, we show that the number of executions of `Type 2(b)` increasing moves in X' is finite.

Label the moves in X' by $1, 2, 3, \ldots$ Let m_1 be a `Type 2(b)` increasing move by process i. Then, we view an increasing move m_2 at another process j as the *cause* of the move m_1 at i, if process j is the last neighbor of i to set its timer value to M before m_1, and move m_2 is the last increasing move by j before m_1 in X' that assigns M to t_j. The move m_1 is thought of as being caused by move m_2. Note that by definition of a `Type 2(b)` increasing moves, $t_j = M$ just before the execution of m_1. Of course, no neighbor of i may have made a move to assign M to its timer in X' prior to the execution of m_1, and some neighbors may have initial timer values of M at the beginning of X'. In this case, we think of the cause of m_1 as an imaginary move $\perp$ at j, where j is an arbitrary neighbor of i with timer value equal to M just before move m_1.

The motivation behind defining a cause of a `Type 2(b)` increasing moves in the above manner is as follows. We know that X' contains only `Type 2(b)` increasing move and decreasing moves. Hence, for every increasing move in X', we can define a cause of the move. Then for any increasing move m_1 by a process i in X', we can follow a backward chain of increasing moves in X' such that each move in the chain is caused by the preceding move in the chain. Since every increasing move has a cause as defined above, such a chain can be constructed until we reach a move that is caused by an imaginary move $\perp$ at some process j. We can also view this chain as starting with an imaginary move $\perp$ at j and and ending with the move m_1 at i such that each move in the chain causes the subsequent move. Using certain properties of such chains, it can be shown that for any two processes i and j, there can be only finitely many chains that begin with an imaginary move $\perp$ at j and end with an increasing move by i. Since there is only finitely many processes with initial timer value M at the beginning of X', this implies that the number of increasing moves by any process i is finite. Therefore, the number of increasing timer moves in X' is finite.

Lemma 5 *The number of executions of* `Type 2(b)` *increasing moves in X' is finite.*

We have thus proved that the total number of increasing moves on the timers is finite. Consider the suffix of the execution sequence after the last increasing timer move by any process. Then, this suffix contains only decreasing timer moves. Clearly, this suffix is finite, since the timer of a process cannot be decremented below 0 by any action of the protocol Q. This leads to the following theorem.

Theorem 2 *The protocol Q terminates in a finite number of steps.*

The following theorem then follows from Theorems 1 and 2.

Theorem 3 *The protocol Q is self-stabilizing.*

5.3.2 Proof of fault-containment

Recall that a timer t_i is said to be consistent if $|t_i - t_j| \leq 1$ for all $j \in N_i$; otherwise, it is inconsistent. A consistent timer with a non-zero value can decrement its value only when its value is equal to or one more than the timer values of all its neighbors. We will refer to decrements of a consistent timer as decrements *in sync* with the neighbors of the process.

The following observation can be easily verified.

Observation 1 *Let t_i be a consistent timer. Then, the set of consistent timers and the set of inconsistent timers in the system remain unchanged after a decrement operation by i.*

Thus, if a timer t_j for an arbitrary process j is consistent before the decrement operation by i, then t_j is consistent after the decrement operation by i. Similarly, if t_j is inconsistent before the decrement operation, then t_j remains inconsistent after the operation.

The proof of fault-containment is organized in three parts. In the first part, we introduce the notion of a *good state* and identify various properties of a good state that are crucial in proving the fault-containment properties of Q. In the second part, using the notion of a good state and its properties, we prove that the fault-gap of Q is $O(M + D)$, where D is the diameter of the network. Finally, in the third part, we show that the containment time of Q is constant and the contamination number of Q is one. Only an overview of the proof is given and the main results are stated. The complete proof appears in [20].

A Good State and Its Properties

We first define a *region* as follows.

Definition 13 A region R is a maximal subset of V such that (i) the graph induced by R is connected, (ii) the timers of all processes in R are consistent, and (iii) at least one process in R has timer value not equal to 0.

Given a region R, the *border* of R is defined as the set B_R of all processes such that $B_R \subseteq V - R$ and every process in B_R is adjacent to some process in R. Thus, a process that is not in a region R or in the border B_R of R cannot be adjacent to any process in R. Note that since R is maximal, no process in B_R has a consistent timer.

A region R is said to be a *closed region* if the timer value of each process in B_R is equal to M. Then, a good state of the system is defined as follows.

Definition 14 A system is said to be in a good state if all regions in the system are closed regions, and the timer value of every process not belonging to any region is 0 or M.

Note that a good state may not have any region at all.

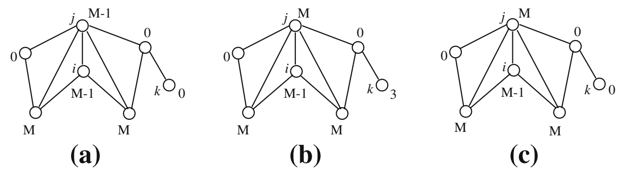

Fig. 6 Examples of region, border, closed region, and good state

Figure 6 shows examples of a region, the border of a region, a closed region, and a good state. The timer values of each process is shown beside it. Note that $n = 7$ and $M > n$. Figure 6a–c shows the same network of processes, but with differing timer values. In all three cases, there is exactly one region in the system consisting of the single process i. The neighbors of i constitute the border of the region. The rest of the processes are not in a region or a border. The state in Fig. 6a is not a good state since j, a process in the border, has a timer value not equal to M, and hence the region is not closed. The state in Fig. 6b is not a good state since k, a process not in any region or border, has a timer value not equal to M. However, note that in this case, the region is closed, since all processes in the border has timer value M. Figure 6c shows an example of a good state. The single region in the system is closed, and all processes not in a region or a border have timer values 0. Note that in this case, other than process i, process k is the only process with a consistent timer. However, k does not form a region by itself since $t_k = 0$ and by definition of a region, at least one process in the region must have a non-zero timer value.

We next identify certain properties of a good state. Recall that a process can execute either a decreasing timer move, or a `Type 1` or `Type 2` increasing timer move. Also recall that a timer t_i of a process i is said to be pseudo-consistent either if t_i is consistent, or if t_i is inconsistent and for every process $j \in \{i\} \cup N_i$, $M - n \leq t_j \leq M$. From the definition of the predicate $decrement(t_i)$ in the protocol Q, we know that t_i can decrement only if t_i is pseudo-consistent.

The following observation can be easily verified.

Observation 2 *In a good state, the timer of any process is either consistent or not pseudo-consistent.*

Thus, in a good state, decrement operations can only be executed by processes with consistent timers. Also, by definition of a good state, the timer of a process not in a region can only be 0 or M. If the timer of a process not in a region is consistent, then the timer value cannot be M, as then the process will belong to a region. Hence, the following observation holds.

Observation 3 *In a good state, if the timer of a process not in a region is consistent, then the value of the timer is* 0.

Hence, a process not in a region cannot decrement its timer, and the following observation holds.

Observation 4 *In a good state, decreasing timer moves can only be executed by processes in a region.*

By definition of a good state, if process j belongs to a border of some region in a good state, then $t_j = M$ and t_j is inconsistent. Since $t_j = M$, all neighbors of j that are in some region must have timer values M or $M - 1$, since all such processes must have consistent timers. Also, since the state is a good state, any neighbor of j not in any region must have timer value 0 or M. Therefore, t_j is inconsistent implies that j must have a neighbor k such that $t_k = 0$, as otherwise, t_j will be consistent. Therefore, no guards are enabled at j and the following observation holds.

Observation 5 *In a good state, a process in the border of some region cannot execute any move.*

The next observation follows from Observation 5, and the fact that a process in a region has a consistent timer, and hence, cannot execute a `Type 2` increasing move.

Observation 6 *In a good state, a* `Type 2` *increasing move can be executed only by a process not in a region or a border.*

Thus, in an arbitrary good state, a process in a region can only execute decreasing moves or `Type 1` increasing moves, a process in a border cannot execute any move, and a process not in a region or a border can execute either `Type 1` or `Type 2` increasing moves.

The following result can be shown.

Lemma 6 *A good state is closed under a decreasing move and a* `Type 2` *increasing move.*

In general, a good state is not closed under a `Type 1` increasing move. However, it can be shown that starting from a 1-faulty state with fault at some process i, only processes in $\{i\} \cup N_i$ can execute a `Type 1` increasing move, and any such execution in a good state results in a good state. This will imply that starting from a 1-faulty state, a good state is closed under all possible moves in the system.

We will now use the above properties of a good state to prove the fault-containment properties of the protocol Q.

Fault-gap of Q

Consider a 1-faulty state f of the system with fault at process i. If the fault at i corrupts only t_i and sets it to 1, then in one round, t_i decrements to 0 and the system reaches a legitimate state. We will not consider this trivial case any more. Hence, in the rest of this proof, we will assume that in the state f, if $t_i = 1$, then G_j is true or $\texttt{Legit}_b(C, j)$ is false for some process $j \in \{i\} \cup N_i$ (that is, the fault at i has also corrupted either the primary portion of the state or some book-keeping variables at i).

Intuitively, the fault-gap of Q is established as follows. We first show (in Theorem 4) that starting from a 1-faulty state with fault at i, within a constant number of rounds, the system reaches a good state with the faulty process i belonging to some closed region R. Consider a process j in B_R. Then, if there is any neighbor of j that is not in R, within one round, all such neighbors will set their timers to M. Intuitively, this will extend the border of the region R by at least one, removing j from B_R and putting it in R. This growing of the region R continues until all processes belong to R (and thus, B_R is empty). It is easy to see that this will happen within $O(D)$ rounds. Thus, after $O(D)$ rounds, all processes have consistent timer values. Thus, any process can make only decreasing moves in sync with its neighbors thereafter, and it is easy to see that within $O(M)$ rounds, all processes will decrement their timers to 0, thereby reaching a legitimate state of Q. Thus the fault-gap of Q is $O(M + D)$.

Starting from the 1-faulty state f, let f' be the first state obtained from f in which some timer value equals M. Note that f' may be the same as f if the fault at i sets t_i to M. Then the following result holds.

Lemma 7 *Starting from f, the state f' is reached within two rounds. Moreover, in f', for exactly one process $j \in \{i\} \cup N_i$, $t_j = M$ and for each process $k \neq j$, $t_k = 0$. Hence, f' is a good state.*

We next argue that any state reachable from f' is a good state. We have already shown in Lemma 6 that a good state is closed under a decreasing move and a `Type 2` increasing move. In general, an arbitrary good state is not closed under a `Type 1` increasing move. However, we will show that any good state reachable from f' is also closed under a `Type 1` increasing move. The following two lemmas identify the processes that can possibly execute a `Type 1` increasing move starting from f.

Lemma 8 *For any process j, if $t_k = M$ for some $k \in \{j\} \cup N_j$ at any state during the execution of Q, then process j cannot execute any `Type 1` increasing move subsequently.*

Lemma 9 *Starting from the 1-faulty state f, a process not belonging to the set $\{i\} \cup N_i$ cannot execute a `Type 1` increasing move.*

Using Lemmas 7 and 8, the following result can be shown.

Lemma 10 *Starting from state f', an execution of a `Type 1` increasing move by a process in $\{i\} \cup N_i$ in a good state results in a good state.*

Lemma 9 and 10 imply that any state reachable from f' is a good state. Moreover, the following result can be easily shown.

Lemma 11 *Starting from f', within two rounds, a state f'' is reached such that $i \in R_1$ for some closed region R_1. Moreover, $t_i = M$ in or before f''.*

The following theorem follows directly from Lemma 7 and Lemma 11.

Theorem 4 *Starting from the 1-faulty state f with fault at i, in constant number of rounds, a state f'' is reached such that f'' is a good state and the faulty process i belongs to some closed region R_1 in f''.*

Since t_i is set to M before reaching f'', the following result holds from Lemmas 8 and 9

Lemma 12 *Starting from state f'', no process can execute a `Type 1` increasing move.*

Hence, any state reachable from f'' is a good state, and in any such state, using Observation 4–6, and Lemma 12, a process in a region can only execute decreasing moves, a process in a border can execute no moves, and a process not in a region or a border can execute only `Type 2` increasing moves. We next show that starting from f'', a state in which all timers in the system are consistent is reached in $O(D)$ rounds.

Any state reachable from f', and hence from f'', is a good state. In any good state, we know from Observation 5 that no process in the border of a region can change state. A move by any process not in a region or border cannot affect a process in a region, since no process in a region is a neighbor of any process not in a region or a border. Since processes in a region can only execute decreasing moves, the following lemma holds from Observation 1.

Lemma 13 *If a process belongs to a region in any state reachable from f'', then it continues to belong to a region in all subsequent states until all timers in the system reaches* 0.

Note that when all timers reach 0, we still have a good state but by our definition of a region, there is no region in the system.

Since a process in a closed region can only execute decreasing moves, the above lemma implies that once a process j belongs to a closed region, it can no longer execute any increasing timer moves.

The following theorem can then be easily proved. The proof shows by induction that after d rounds, all processes at shortest distance d from i are included in the closed region containing i. Since i initially belongs to a closed region within a constant number of rounds, all processes are included in a single closed region in $O(D)$ rounds. The border of this region is obviously empty, and thus all processes have consistent timer values.

Theorem 5 *Starting from state f'', a state in which all processes have consistent timer values is reached in $O(D)$ rounds.*

Finally, we show that starting from a state in which all timers are consistent, a legitimate state is reached in $O(M)$

rounds. The proof follows from the fact that at each round, the processes with the current maximum timer value must decrease their timer by one.

Theorem 6 *Consider a state s' reachable from f'' in which all processes have consistent timer values. Then, starting from s', a legitimate state is reached within M rounds.*

The following theorem follows from Theorems 4–6.

Theorem 7 *The fault-gap of Q is $O(M + D)$.*

Containment Time and Contamination Number of Q

We will next prove that the containment time of Q is constant and the contamination number of Q is one. We will continue to use f to denote a 1-faulty state with fault at process i and f' to denote the first state reached from f in which some timer in the system has a value of M. We have seen in Lemma 7 that in f', for exactly one process $j \in \{i\} \cup N_i$, $t_j = M$, and for any process $k \neq j$, $t_k = 0$.

The following result can be easily proved.

Lemma 14 *Starting from the 1-faulty state f, the timer of every process is set to M exactly once before a legitimate state is reached.*

The following lemma shows that all decrements of the timers must be in sync.

Lemma 15 *Starting from the 1-faulty state f, any process k decrements t_k only if $0 \leq t_k - t_\ell \leq 1$ for all $\ell \in N_k$.*

The above two lemmas show that all processes will set their timers to M once, and then decrement it down to 0 in synchrony with their neighbors. That is, all decrements of a process are in synchrony with their neighbors. Since the steps of the protocol C are executed along with the decrements of the timer, this implies that no process executes a step of C until all its neighbors have finished executing the previous step of C. Thus all processes will execute the protocol C synchronously. Since by the fault-containment property of C, only process i and its neighbors can change state before $\mathtt{Legit}_p(Q)$ is established, the following lemma follows.

Lemma 16 *Starting from the state f', the predicate $\mathtt{Legit}_p(Q)$ holds when $t_i \leq M - c$ and for all $j \in N_i$, $t_j \leq M - c$.*

The following lemma bounds the time needed for the timers of i and any $j \in N_i$ to reach $M - c$.

Lemma 17 *Starting from the state f', a state in which $t_i \leq M - c$ and $t_j \leq M - c$ for all $j \in N_i$ is reached within $2c + 2$ rounds.*

Since f' is reached from a 1-faulty state within a constant number of rounds, Lemma 16 and Lemma 17 shows that starting from a 1-faulty state, the predicate $\mathtt{Legit}_p(Q)$ is established within a constant number of rounds. Hence, the containment time of Q is constant. By definition of C, only the faulty process i can change the primary portion of the state before $\mathtt{Legit}_p(Q)$ is established. Also, after $\mathtt{Legit}_p(Q)$ is established, no process can change the primary portion of its local state. Hence the contamination number of Q is one. Hence the following theorem holds.

Theorem 8 *The containment time of Q with respect to $\mathtt{Legit}_p(Q)$ is $O(1)$, and the contamination number of Q is 1. Hence, Q is a fault-containing protocol.*

5.4 Implementation of the protocol C

We finally present two alternate implementations of the synchronous protocol C described in the previous section. The first implementation has a greater space overhead than the second, but it requires only 3 synchronous steps to establish $\mathtt{Legit}_p(Q)$ when started in a 1-faulty state. The second implementation is computationally not as efficient, but it has a space overhead of $O(1)$. The proof of correctness of the implementations appear in [20].

5.4.1 First implementation

Without loss of generality, assume that any process i uses a single variable[2] x_i in P. Thus G_i, is a predicate defined over x_i and x_j for all $j \in N_i$. Each process i in protocol C, in addition to x_i, has one additional variable $s_i[j]$ for each $j \in N_i$. The variable $s_i[j]$ is meant to contain a copy of the variable x_j. Consider a 1-faulty state with fault at process i that corrupts arbitrarily one or more of the variables belonging to the set $\{x_i\} \cup \{s_i[j] \mid j \in N_i\}$. Then, protocol C establishes $\mathtt{Legit}_p(Q)$ within 3 synchronous steps, with only i changing its state. If $|N_i| > 1$, the fault at i is corrected in the first two steps. If $|N_i| = 1$, the fault is corrected in the third step.

For notational convenience, we define the following two predicates:

$$no\_P\_fault(i, y) \equiv (\forall j \in N_i : s_j[i] = y) \wedge (x_i = y)$$
$$P\_fault(i, y) \equiv (\forall j \in N_i : s_j[i] = y) \wedge (x_i \neq y)$$

The predicate $no\_P\_fault(i, y)$ indicates that the variable x_i with value y is consistent with the copies kept in the neighboring processes. The predicate $P\_fault(i, y)$ indicates that the copies of the variable x_i in the neighboring processes are all equal to y, but the actual value of the variable x_i is not equal to y. Hence, if $P\_fault(i, y)$ is true, it is possible that a fault at i has corrupted x_i. Note that a process i may satisfy

[2] Multiple variables at a process i can simply be thought of as multiple fields of x_i.

neither of the two predicates for any value of y. However, both the predicates cannot be satisfied at the same time. The protocol C is shown in Fig. 7. A brief description of the three steps follows.

If the fault has corrupted x_i, let the value of x_i before the fault be y. Then $P\_fault(i, y)$ will be true and $no\_P\_fault(i, y)$ will be false. If $|N_i| > 1$, step 1 will correct this fault. If $|N_i| = 1$, step 3 will correct the fault. In both cases, any inconsistencies in the copies of neighbors' variables kept at i are also corrected. No other processes change their state. However, it is possible for a fault at i to corrupt only the variables $s_i[j]$ for one or more $j \in N_i$. In this case, we may enter the following scenario. Assume that $|N_i| > 1$, a variable $s_i[j]$ is corrupted for some j, and $|N_j| = 1$. If $s_i[j] = y_1$ for some y_1 after the fault, then $P\_fault(j, y_1)$ is true. This scenario is shown in Fig. 8. In this case, it is necessary for the variable $s_i[j]$ to be corrected before step 3 is entered, as otherwise j will wrongly change x_j in step 3. Step 2 incorporates this correction. When an inconsistency in one or more copy variables is detected, process i tries to determine if the fault is at i or if the fault is at some neighbor of i. In the latter case, process i will not take any action, and let the neighbor detect and repair the fault. The *NO-CORRECTION condition* identifies scenarios in which the fault is at some neighbor of i, and hence, process i takes no action. If $s_i[j]$ is not the same as x_j for more than one neighbor j of i, the fault must be at i, and i corrects all inconsistent copies. However, if $s_i[j]$ is inconsistent for exactly one neighbor j, the inconsistency may be due to the corruption of $s_i[j]$ at i, or the corruption of x_j at j. However, if $|N_j| > 1$, the fault cannot be at j, since then, process j would have corrected x_j during step 1. Hence, i again corrects $s_i[j]$. However, if $|N_j| = 1$, it is harder to determine if the fault is at i or at j. If after setting x_j to $s_i[j]$ (that is, assuming that the fault is at j), at least one of G_i and G_j is true, then surely the fault cannot be at j, as the state before the fault was a legitimate state in which G_x was false for every process x. Hence, i again corrects $s_i[j]$. However, if setting x_j to $s_i[j]$ makes both G_i and G_j false, i assumes that the fault is at j and allows j to correct x_j during step 3. Note that since i is the only neighbor of j (since, $|N_j| = 1$), it suffices to check only the status of G_i and G_j.

```
Step 1:
  if ($|N_i| > 1$) and ($\exists y : P\_fault(i, y)$) then
      $x_i := y$;
      for all $j \in N_i$: if ($s_i[j] \neq x_j$) then $s_i[j] := x_j$.
  Create a new variable $n_i$ and set it to $|N_i|$.

Step 2:
  if ($|N_i| > 1$) and ($\exists y : no\_P\_fault(i, y)$)
                              and ($\exists j \in N_i : s_i[j] \neq x_j$)
  then
      if ($s_i[j] \neq x_j$) for exactly one $j \in N_i$ and $n_j = 1$
           and changing $x_j$ to $s_i[j]$ will make both
                                   $G_i$ and $G_j$ false
           // "NO-CORRECTION condition" //
      then
         do nothing
      else
         for all $j \in N_i$: if ($s_i[j] \neq x_j$) then $s_i[j] := x_j$;

Step 3:
  for all processes $i$, deallocate $n_i$.
  if $|N_i| = 1$ then
      if ($\exists y : P\_fault(i, y)$) then $x_i = y$;
      if $N_i = \{j\}$ and $s_i[j] \neq x_j$ then $s_i[j] := x_j$;
```

Fig. 7 An implementation of C that requires three synchronous steps

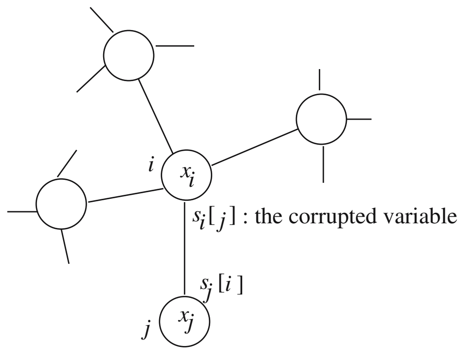

Fig. 8 A scenario in which the corruption of a single copy variable at one process i can cause an incorrect change in another process j

Note that the protocol requires process i to compute the value of G_j when $|N_j| = 1$ in step 2. Since i is the only neighbor of j, i can check the status of G_j assuming that i has access to the program executed by j. This assumption is made here for simplicity of the protocol, but is not necessary. The status of G_j can be known by i using an additional temporary boolean variable that is set appropriately by a node j with $|N_j| = 1$ in step 1 if $x_j \neq s_i[j]$.

The above protocol will correct a single fault in all cases except when the network is a just a single edge. In such a trivial case, if $\texttt{Legit}_p(Q)$ is not true after execution of C, the protocol P, executed as part of Q, will establish $\texttt{Legit}_p(Q)$. The implementation of C presented here can be modified to work correctly for a single edge also. However, we do not do so here since the complexity of P in this case will typically be trivial.

Note that when this particular implementation of C is used in Q, the predicate $\texttt{Legit}_b(C, i)$ for a process i is defined as follows.

$$\texttt{Legit}_b(C, i) \equiv (\forall j \in N_i : s_i[j] = x_j)$$

That is, the copies that i keeps of the x variables of its neighbors are all consistent with the actual value of the x variables in the neighbors. Then, $\texttt{Legit}_b(C) \equiv \forall i : \texttt{Legit}_b(C, i)$. Also, the book-keeping action that a process i executes when the timer t_i is in the range $[2..b]$ is simply

$$\forall j \in N_i : s_i[j] = x_j.$$

Thus, the value of b in this case is simply 2, and a process i executes the book-keeping operation when it decrements its timer from 2 to 1. Note that after the timer of a process i has reached $b = 2$, $\texttt{Legit}_p(Q)$ is true by the assumption on the value of M and hence, the variable x_i can no longer change. Hence, the copies of x_i made by the neighbors of i when their timers decrement from 2 to 1 (and hence, t_i must be 2 or less) are the correct copies of x_i. Thus, $\texttt{Legit}_b(C)$ is established when all timers are decremented to 1.

5.4.2 An alternate implementation of C

In this alternate implementation of C, we give up computational efficiency to an extent to reduce the space overhead of achieving fault-containment. In particular, in this alternate implementation of C, book-keeping variables have temporary existence — they are created in the first synchronous step of C and are destroyed in the last step. Because of this, C contributes $O(1)$ to the space overhead of Q, since in a legitimate state of Q, none of the book-keeping variables of C are in existence. A brief sketch of this alternate implementation of C is given below. We describe each step separately and provide occasional comments to give intuition. For an integer $k \geq 0$ we use the notation N_i^k to denote the set of processes that are at a distance of k from i.

Step 1 Each process i allocates space for a local variable $s_i[0]$ and assigns to $s_i[0]$ the set $\{x_i\}$. Recall the assumption we made at the outset of Sect. 5.4.1 that each process i uses a single variable x_i in P.

Steps 2–5 In Step k, $k = 2, 3, 4, 5$, each process i allocates space for a variable $s_i[k]$ and assigns to $s_i[k]$ the union of the sets $s_j[k-1]$ for all $j \in \{i\} \cup N_i$.

Comment: After Step 5, each process i has a local variable $s_i[4]$ that contains a complete snapshot of the portion of the network that is within distance 4 from process i.

Step 6 Each process i computes the predicate $\texttt{oneFaulty}_i$ defined as

$$(G_i \vee \exists j \in N_i : G_j) \wedge (\forall j \in N_i^2 \cup N_i^3 : \neg G_j)$$

Comment: In a 1-faulty state in which i is a faulty process, $\texttt{oneFaulty}_i$ is true. It is possible that even if i is not a faulty process in a 1-faulty state, $\texttt{oneFaulty}_i$ is true. However, it is easy to verify that in a 1-faulty state, if $\texttt{oneFaulty}_i$ and $\texttt{oneFaulty}_j$ are both true for some $i \neq j$, then distance between i and j is at most 2. To evaluate $\texttt{oneFaulty}_i$, process i has to evaluate guards at itself, its neighbors, and at processes that are at a distance 2 or 3 away and hence it needs to know information about processes that are at a distance of at most 4. Since this information is available in the variable $s_i[4]$, process i is able to compute $\texttt{oneFaulty}_i$. Note that processes may need unique identifiers, at least within the subgraph of radius 3 around i, for i to be able to distinguish between processes in N_i and processes in $N_i^2 \cup N_i^3$.

Step 7 Each process i for which $\texttt{oneFaulty}_i$ is true, computes the predicate

$$\texttt{canRepair}_i \equiv \exists \alpha : \texttt{canRepair}_i(\alpha)$$

where $\texttt{canRepair}_i(\alpha)$ is defined as

if the local state of process i is changed to α then G_i is false and for all $j \in N_i$, G_j is false.

Comment: In a 1-faulty state $\texttt{oneFaulty}_i \wedge \texttt{canRepair}_i$ is true if and only if i is faulty. Recall that a 1-faulty state is a state that differs from some legitimate state in the local state of exactly one process. It is possible that a 1-faulty state differs from one legitimate state in the local state of a process i and from some other legitimate state in he local state of process j, where i and j are distinct. So several processes can be considered the only faulty processes in a 1-faulty state and any one of these could repair the fault. Hence the only problem to be solved now is to ensure that exactly one of the faulty processes changes its local state and repairs the fault. Otherwise, if several faulty processes change their local state simultaneously, the fault may not be repaired. So processes have to negotiate to pick one process that fixes the fault. This task is performed in the next few steps.

Step 8 Each process i allocates space for a variable $t_i[0]$. If $\texttt{oneFaulty}_i \wedge \texttt{canRepair}_i$ is true, then process i assigns the set $\{i\}$ to $t_i[0]$; otherwise the empty set is assigned to $t_i[0]$.

Step 9-10 In Step k, $k = 9, 10$, each process i assigns to $t_i[k-7]$, the union of $t_j[k-8]$ for all $j \in \{i\} \cup N_i$.

Comment: At the end of Step 10, each process has a local variable $t_i[2]$ that contains the set of all processes j at distance no more than 2 from i such that $\texttt{oneFaulty}_j \wedge \texttt{canRepair}_j$ is true. As mentioned in an earlier comment, if for some $i \neq j$, $\texttt{oneFaulty}_i$ and $\texttt{oneFaulty}_j$ are both true, then distance between i and j is at most 2. Hence, in a 1-faulty state, if $\texttt{oneFaulty}_i \wedge \texttt{canRepair}_i$ is true, then $t_i[2]$ contains the set of *all* processes j for which $\texttt{oneFaulty}_j \wedge \texttt{canRepair}_j$ is true.

Step 11 A process i such that $i = \min(t_i[2])$, changes its local state to α where $\texttt{canRepair}_i(\alpha)$ is true.

Comment: The process with minimum identifier among all faulty processes in a 1-faulty state is elected to fix the fault.

Step 12 Each process i deallocates space for all variables $s_i[k]$, $0 \leq k \leq 4$, and $t_i[k]$, $0 \leq k \leq 2$.

Discussion Since the above implementation of C requires 12 steps, the value of c is 12. Recall that $[M - c + 1, M]$

is the subrange of timer values in which the protocol C executes. Since there are no book-keeping variables that remain in existence after C has finished execution, $\texttt{Legit}_b(C, i)$ is simply defined to be true. Since there are no book-keeping variables, there are no book-keeping actions to be performed and therefore the range $[2..b]$ within which such actions are performed, can be shrunk to 0.

The problem with the above implementation of C is that it is not clear how to compute the predicate $\texttt{canRepair}_i$ in general. For many protocols, if a process i is faulty in a 1-faulty state then G_i is true and the execution of the corresponding action A_i repairs the fault. So in such cases, the protocol itself is an excellent guide on how to compute $\texttt{canRepair}_i$. This pleasant situation does not hold for all protocols; for example consider the spanning tree protocol described in Sect. 2. In the 1-faulty state shown in Fig. 1(c) process x is faulty, in fact, it is the only process that has an enabled guard. But, none of the actions of process x as prescribed by the spanning tree protocol, can repair the fault. This particular problem can be easily solved by adding a new action to the protocol. But, in general appropriate new actions that can be added to protocols without affecting their self-stabilizing properties may not be easy to derive. So the above implementation of the protocol C works only for protocols for which the predicate $\texttt{canRepair}_i$ can be computed efficiently.

The first implementation of C did not have this problem since the old state was memorized. Hybrid versions of C in which the old state is partly memorized and partly computed might alleviate both the problems of space overhead and the computational difficulty of the above solution.

6 Conclusion

In this paper, we have formally introduced the notion of fault-containment in the context of self-stabilization. We have specified metrics to evaluate the fault-containment properties of a self-stabilizing protocol, and formally defined a fault-containing self-stabilizing protocol in terms of them. In the main result of our paper, we present a transformer that can automatically convert any non-reactive self-stabilizing protocol into an equivalent fault-containing self-stabilizing protocol. We believe that the results presented here are an important step towards making self-stabilizing protocols more practical.

The definitions presented in Sect. 4 can be naturally extended to specify fault-containment from multiple faults. However, the transformer presented in Sect. 5 generates protocols that provide fault-containment from single faults only, since the component protocol C recovers only from a single transient fault. If we extend C so that it can efficiently recover from a k-faulty state for any $k \geq 1$, then using the same timer-based approach we can design a transformer that can generate self-stabilizing protocols that provide fault-containment from multiple faults. It can be shown that for a synchronous system, the transformer presented in this paper can produce self-stabilizing protocols that recover from multiple faults with the same guarantees on fault-gap and containment time, provided that the shortest distance between any two faults in the network is at least four. In this case, it can be easily shown using the same proof technique used in the single fault case that a good state is formed within a constant time, with each faulty node belonging to a closed region. The proof then proceeds similarly. The faults in this case are sufficiently dispersed around the network so as to not affect each other.

The protocols generated by the transformer has a constant containment time, meaning that the observable effects of a single fault is repaired very fast. However, the fault-gap of the protocol can be high, since the value of M may be large. It might be possible to reduce the fault-gap of the protocol using the following approach. Suppose that M is not as large as required by our transformer. In this case, it is possible that a process can decrement its timer from M to 0, but the system may not have reached a legitimate state. The process might then reset its timer to M, and start decrementing down to 0 again. Depending on the value of M, this process may be repeated a number of times. Note that each time the process decrements from M to 0, it first executes the protocol C. So if we can ensure that the repeated execution of the steps of C within the execution of P do not affect progress towards self-stabilization, then the protocol will reach a legitimate state eventually, Thus, if we use an implementation of C satisfying this condition, we can generate a class of transformers, one transformer for each value of M. Reducing the value of M reduces the fault-gap of the generated protocol, but increases the stabilization time, since the steps of C have to be performed multiple times. So this approach promises to yield a sequence of transformers that reveals a trade-off between fault-gap and stabilization time. We are currently investigating if the implementation of C presented in this paper satisfies this property.

Acknowledgements We thank the referees for their valuable comments and suggestions, which improved both the content and the presentation of the paper considerably.

References

1. Afek, Y., Dolev, S.: Local stabilizer. In: Proceedings of the 5th Israeli Conference on Theory of Computing and Systems, pp. 74–84 (1997)
2. Anagnostou, E., El-Yaniv, R., Hadzilacos, V.: Memory adaptive self-stabilizing protocols. In: WDAG92 Distributed Algorithms 6th International Workshop Proceedings. LNCS vol. 647, pp. 203–220. Springer, Heidelberg (1992)

3. Arora, A., Gouda, M.G.: Distributed reset. IEEE Trans. Comput. **43**, 1026–1038 (1994)
4. Arora, A., Kulkarni, S.S.: Designing masking fault-tolerance via nonmasking fault-tolerance. In: SRDS95 Proceedings of the 14th Symposium on Reliable Distributed Systems, pp. 174–185 (1995)
5. Arora, A., Kulkarni, S.S.: Component based design of multitolerance. IEEE Trans. Softw. Eng. **43**, 1026–1038 (1997)
6. Awerbuch, B.: Complexity of network synchronization. J. ACM **32**, 804–823 (1985)
7. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 652–661 (1993)
8. Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In: Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, pp. 258–267 (1991)
9. Beauquier, J., Genolini, C., Kutten, S.: Optimal reactive k-stabilization – the case of mutual exclusion. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, pp. 209–218 (1999)
10. Bruell, S.C., Ghosh, S., Karaata, M.H., Pemmaraju, S.V.: Self-stabilizing algorithms for finding centers and medians of trees. SIAM J. Comput. **29**(2), 600–614 (1999)
11. Buskens, R.W., Bianchini, R.P.: Self-stabilizing mutual exclusion in the presence of faulty nodes. In: FTCS95 Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems, pp. 144–153 (1995)
12. Chen, N.S., Yu, H.P., Huang, S.T.: A self-stabilizing algorithm for constructing spanning trees. Informat. Process. Lett. **39**, 147–151 (1991)
13. Couvreur, M., Francez, N., Gouda, M.G.: Asynchronous unison. In: Proceedings of the 12th International Conference on Distributed Computing Systems, pp. 486–493 (1992)
14. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Communi. ACM **17**, 643–644 (1974)
15. Dolev, S.: Optimal time self-stabilization in dynamic systems. In: WDAG93 Distributed Algorithms 7th International Workshop Proceedings. LNCS, vol. 725, pp. 160–173 Springer, Heidelberg (1993)
16. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
17. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, pp. 3.1–3.15 (1995)
18. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. In: WDAG91 Distributed Algorithms 5th International Workshop Proceedings. LNCS, vol. 579, pp. 167–180 Springer, Heidelberg (1991)
19. Ghosh, S., Gupta, A.: An exercise in fault-containment: Self-stabilizing leader election. Informat. Process. Lett. **5**(59), 281–288 (1996)
20. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing distributed protocols. Technical Report TR-00-01, Department of Computer Science, The University of Iowa, Iowa City (2000)
21. Ghosh, S., Gupta, A., Pemmaraju, S.V.: A fault-containing self-stabilizing algorithm for spanning trees. J. Comput. Informat. **2**, 322–338 (1996)
22. Ghosh, S., Gupta, A., Pemmaraju, S.V.: Fault-containing network protocols. In: Proceedings of 12th Annual ACM Symposium on Applied Computing (1997)
23. Ghosh, S., Gupta, A., Pemmaraju, S.V.: A self-stabilizing algorithm for the maximum flow problem. Distributed Comput. **10**, 167–180 (1997)
24. Ghosh, S., He, X.: Scalable self-stabilization. In: Proceedings of the 4th Workshop on Self-Stabilization, pp. 18–24 (1999)
25. Gopal, A.S., Perry, K.J.: Unifying self-stabilization and fault-tolerance. In: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing, pp. 195–206 (1993)
26. Gouda, M.G., Schneider, M.: Maximum flow routing. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, pp. 2.1–2.13 (1995)
27. Herman, T.: Superstabilizing mutual exclusion. In: Proceedings of 1st International Conference on Parallel and Distributed Processing: Techniques and Applications (1995)
28. Kutten, S., Patt-Shamir, B.: Asynchronous time-adaptive self-stabilization. In: Brief Announcement, Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (1998)
29. Kutten, S., Patt-Shamir, B.: Stabilizing time-adaptive protocols. Theor. Comput. Sci. **220**, 93–111 (1999)
30. Kutten, S., Peleg, D.: Fault-local distributed mending. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 20–27 (1995)
31. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
32. Nelson, V.: Fault-tolerant computing: fundamental concepts. IEEE Comput., pp. 19–25 (1990)
33. Parlati, G., Yung, M.: Non-exploratory self-stabilization for constant-space symmetry-breaking. In: Algorithms ESA 94. LNCS, vol. 855, pp. 183–201 Springer, Heidelberg (1994)
34. Schneider, M.: Self-stabilization. ACM Comput. Surveys **25**, 45–67 (1993)