2010 International Conference on Distributed Computing Systems

Fault-containing self-stabilization in asynchronous systems with constant fault-gap

Sven Köhler, Volker Turau
Institute of Telematics
Hamburg University of Technology, Hamburg, Germany
{sven.koehler, turau}@tu-harburg.de

Abstract

This paper presents a new transformation which adds fault-containment properties to any silent self-stabilizing protocol. The transformation features a constant slow-down factor and the fault-gap – that is the minimal time between two containable faults – is constant. The transformation scales well to arbitrarily large systems and avoids global synchronization.

Keywords

fault-containment; self-stabilization; single transient faults

1. Introduction

The notion of self-stabilizing was coined by E. W. Dijkstra [6]. A self-stabilizing distributed system provably converges to a set of legitimate states in finite time, regardless of its initial state and without any external intervention. The work of Dijkstra was then picked up by L. Lamport who called it a "milestone in work on fault-tolerance" [16]. The system state after a transient fault can simply be seen as another initial state. The self-stabilizing system is guaranteed to converge again. This makes self-stabilization an elegant and formal approach for non-masking tolerance of transient faults.

However, it is a well known problem that even a small scale fault can cause disruption of large parts of the system and that it may take a rather long time, until the system reaches a legitimate state again. This is mostly due to the fact that each node of the system can only access local data. This makes it hard for a node to identify faulty incoming data and thus faulty data can spread from node to node. This process is called contamination.

As an example, consider a protocol that establishes a shortest path spanning tree. It maintains two variables on each node. One to store the distance to the designated root node and another variable with a pointer to a neighboring node. Each node computes its own distance variable by looking at the distance variables of its neighbors. The pointer variable simply points to the neighbor with the lowest distance. After stabilization, the pointers form a spanning tree.

As it happens, a node with a high distance to the root node is affected by a fault. Its distance variable has been set to a very low value by the corruption. To other nodes, this will appear like a shortcut to the root node has been discovered. Depending on the choices of the scheduler, the false information may spread throughout most of the system before the faulty node repairs its variable. Hence, at some point during the stabilization process, many of the pointer variables will point in direction of the faulty node, rather than the direction of the real root.

If the spanning tree induced by the pointer variables is used for routing, then such a disruption can cause loss of a large number of packets. It is very desirable that the effects of such faults are restrained in space and time. Depending on the application, one might be particularly interested in a quick recovery of only the pointer variables whereas the other variables can be allowed to fluctuate until a legitimate state is reached again.

Fault-containment promises to lower the impact of small scale faults. It has been put in the context of self-stabilization for the first time by Ghosh et al. [8]. Fault-containing self-stabilizing protocols combine two aspects: First, such protocols recover quickly from small scale faults. Secondly, they also recover from large scale faults because they are self-stabilizing. A further motivation is the fact that small scale faults are usually much more frequent than large scale faults. Handling them efficiently can greatly increase the availability of the system.

Ghosh et al. also defined metrics to measure the fault-containment qualities of self-stabilizing protocols. The most significant two are containment time and contamination number. The former describes the impact of a fault in time, namely how long it takes after a fault, until the output of a protocol can be considered to be correct again. The latter describes the impact of a fault in space, namely how many nodes change their output variables and thus can be expected to stop working temporarily. Furthermore, there is the fault-gap. It indicates, how frequent small scale faults can occur. A fault-containing protocol is only guaranteed to handle two subsequent faults efficiently, if the interval between them is no smaller than the fault-gap. It is usually greater than the containment time. After the output variables have been restored, the protocol may have to prepare for the containment of another fault.

The main contribution of this paper is a transformation that maps any silent self-stabilizing protocol P to a fault-containing self-stabilizing protocol P_F. The containment time,

This research was funded by the German Research Foundation (DFG), contract number TU 221/3-1.

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.10

IEEE computer society

contamination number, and also the fault-gap of protocol P_F are constant. The fault-gap is significantly lower than that of all known general transformations for asynchronous systems. In addition, the transformation is shown to preserve the stabilization time of protocol P, except for a constant slow-down factor.

2. Model of Computation

A distributed system is represented as an undirected graph (V, E) where V is is the set of *nodes* and $E \subseteq V \times V$ is the set of *edges*. The topology is assumed to be fixed. If two nodes are connected by an edge, then they are called *neighbors*. The set of neighbors of node v is denoted by $N(v) \subseteq V$. The number of nodes in the system is denoted by $n = |V|$ and Δ denotes the maximal degree of the nodes.

Each node stores a set of variables. The values of all variables constitute the *local state* of a node. Let σ denote the set of local states of a node. The *global state* of the system is the vector of all local states in the system and $\Sigma = \sigma^n$ denotes the set of global states.

Nodes communicate via locally shared memory. Every node can read the variables of all its neighbors. Write access is prohibited. Each node $v \in V$ executes a protocol consisting of a list of rules. Each rule consists of a *guard* and a *statement*. A guard is a Boolean expression over the variables of node v and its neighbors. A rule is called *enabled* if its guard evaluates to true. A node is called *enabled* if one of the rules is enabled.

The execution of the statements is controlled by a scheduler. It operates in *steps*. At the beginning of step i, it first non-deterministically selects a non-empty subset $S_i \subseteq V$ of enabled nodes. Each node in S_i then executes the statement of the enabled rule. This is called a *move*. A step is finished, if all nodes have finished their moves. The sequence $\langle S_1, S_2, S_3, \ldots\rangle$, $S_i \subseteq V$ is called *schedule*. This paper focuses on the *central daemon scheduler* which selects exactly one node during each step ($|S_i| = 1$). No assumptions on the fairness of the scheduler are made.

An *execution* $\langle c_0, c_1, c_2, \ldots\rangle$, $c_i \in \Sigma$ is a sequence of global states c_i where c_0 is the *initial state* and c_i is the global state of the system after the i-th step. In other words, if the system is in the state c_i and all nodes in S_{i+1} execute a move, then this yields c_{i+1}.

The run-time of a protocol is measured in *rounds*. Let x be an execution and $x_0 = x$. Then x is partitioned into rounds by induction over $i = 0, 1, 2, \ldots$: round r_i is defined to be the minimal prefix of x_i, such that each node $v \in V$ has either executed a move or is disabled at least once within r_i. The execution x_{i+1} is obtained by removing the prefix r_i from x_i. The intuition is that within a round, each node that is enabled at the beginning of the round, gets the chance to execute a move if it has not become disabled by a move of its neighbors.

2.1. Self-Stabilization

Let P be a protocol and let $Legit_P$ denote a Boolean predicate over the set of global states Σ. If $Legit_P(c)$ is true, then the global state c is called *legitimate*. Protocol P is said to be *self-stabilizing* with respect to $Legit_P$, if both the following properties are satisfied. *Convergence*: for any execution of protocol P, a legitimate state is reached within a finite number of steps. *Closure*: for any execution of protocol P it holds that once a legitimate state is reached, all subsequent states are also legitimate. Furthermore, a self-stabilizing protocol P is called *silent*, if it terminates after a finite number of steps. A protocol has terminated, if all nodes are disabled.

2.2. Fault-Containment

A global state that can be derived from a legitimate state by perturbing the variables of a single node is called 1-*faulty*. To efficiently recover from such global states, the transformation of protocol P into protocol P_F adds a set of new variables to each node. These variables are called *secondary*. The variables of the original protocol P are called *primary*. By this distinction of primary and secondary variables, any global state is split up into an ordered pair of a primary state and a secondary state.

Let $Legit_P$ denote the predicate that decides whether the primary state is legitimate with respect to protocol P and let protocol P_F be self-stabilizing with respect to the predicate $Legit_{P_F}$. Protocol P_F is required to satisfy the following:

1) For any legitimate state of protocol P_F, the primary state is also legitimate. $\forall c : Legit_{P_F}(c) \Rightarrow Legit_P(c)$
2) For any execution of protocol P_F that starts in a 1-faulty state the following property holds: After the first legitimate primary state all subsequent primary states are also legitimate. *

Starting in a 1-faulty state, the *containment time* denotes the number of rounds needed to reach a legitimate primary state ($Legit_P$ holds). The *contamination number* denotes the number of nodes that change their primary variables during that time. The *fault-gap* denotes the number of rounds that the system needs to reach a fully legitimate state ($Legit_{P_F}$ holds).

Protocol P_F is called *fault-containing* with respect to $Legit_P$, if both worst-case containment time and worst-case contamination number are constant.

2.3. Nonstandard Extensions

Without loss of generality, it is assumed that per node only one rule of a protocol can be enabled at a time. Then it is possible to define a guard G_X and statement S_X, such that protocol X can be written as a single rule. The guard G_X is the disjunction of all guards of protocol X. The statement S_X simply tests all guards again, and executes the statement of the enabled rule.

Furthermore, a notation for the evaluation of a guard for a given node v, topology, and specific local states is needed.

*. This differs from the definition in [8]. There, this is required to hold for any execution of protocol P_F, regardless of the initial state. However, we do not know of any transformation which satisfies this requirement.

In agreement with the above, the predicate $G_X(v)$ is defined to be true if and only if protocol X is enabled on node v in the current topology and configuration. The predicate $G_X(v : x, u : y)$ is defined to be true if and only if protocol X is enabled on node v in a virtual topology and configuration. In the virtual topology, v and u are the only nodes and u is the only neighbor of v. In the virtual configuration, the local states of nodes v and u equal x and y respectively (also see the end of Section 4.2.2).

To run multiple protocols on one node, a naive composition is used. It is called *combining composition* and is denoted by $A \circ B$. The protocol $A \circ B$ consists of a single rule only. The guard is defined as $G_A \vee G_B$ and the statement consists of the two commands **if** G_A **then** S_A and **if** G_B **then** S_B. Note that the guard G_B is to be re-evaluated after the execution of S_A. The statement S_A may change variables that affect G_B.

3. Related Work

Ghosh et al. [8], [9] present the first transformer that maps any silent self-stabilizing protocol P to a silent fault-containing self-stabilizing protocol P_F. They describe two methods for fault repair. The first method increases the space required on each node by a factor of about $\Delta + 1$. The second method increases the requirements by a factor in the order of Δ^4, but only during the repair phase. Independent of the chosen repair method and with all given optimizations applied, the fault-gap is still $\Omega(n)$. Mainly because a fault has an impact on the secondary variables of the whole system. In addition, an upper bound of the system size must be known a priori. Making the transformer adjust to the size of the system at run-time seems to be a non-trivial task.

In [8], Ghosh et al. contribute an important impossibility result. It is shown that it is impossible for a transformer to map any self-stabilizing protocol P with stabilization time T to a fault-containing self-stabilizing protocol P_F that stabilizes within time $T + \mathcal{O}(1)$ and has a fault-gap of $\mathcal{O}(1)$. The main contribution of our paper is a transformer with fault-gap $\mathcal{O}(1)$ and stabilization time $\mathcal{O}(T)$.

Another way of creating fault-containing protocols is explored by Ghosh and He [12]. They introduce the model of priority scheduling and construct a spanning tree protocol to illustrate that the new model helps designing fault-containing protocols. The protocol has a constant fault-gap which is shown to be preserved under a transformation that adapts priority scheduling to the central daemon scheduler. But it remains an open question, whether priority scheduling can be used to add fault-containment to arbitrary protocols.

Dasgupta et al. investigate a probabilistic variant of fault-containment. They apply the technique to the persistent-bit problem [4] and leader election [5]. The technique seems to be suitable for weakly-stabilizing systems [5] only. Again, it is an open question, whether the technique can be extended to a general transformation.

Herman and Pemmaraju [13] try to avoid any replication overhead. They use error-detecting codes to add fault-containment to self-stabilizing protocols. Faults are recognized and contained with high probability. The transformation applies to certain classes of protocols only. If a fault is contained, then this happens within a single step.

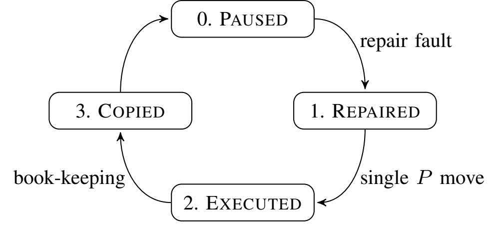

Figure 1. States of a node and the transition-actions

Furthermore, many problem specific fault-containing self-stabilizing protocols have been designed: Maximal Independent Set [17], Spanning Tree [10], [11], [14], and Leader Election [7]. Problem specific protocols usually require less space and/or stabilize faster than solutions based on a general transformation. Yamauchi et al. [18], [19] show that more complex fault-containing self-stabilizing protocols can be obtained by hierarchical composition of existing fault-containing protocols, without loosing their fault-containment properties.

Concepts similar to fault-containing self-stabilization are time-adaptiveness and k-strongness. Time-adaptive self-stabilization is introduced by Kutten and Patt-Shamir [15]. The recovery time of such protocols is linear in the number of faults. Burman et al. [3] published the first time-adaptive and fully self-stabilizing protocol for the majority consensus problem, a variation of the persistent bit problem. k-strong self-stabilization is introduced by Beauquier et al. [2]. Apart from other properties, a k-strong self-stabilizing protocol stabilizes from a k-faulty state within $\mathcal{O}(k)$ time which is similar to time-adaptiveness. A transformation for asynchronous protocols is given. The transformation is rather complex and the space requirements increase by a factor in the order of 3Δ. In addition, the resulting 1-strong protocol does not work in asynchronous systems.

4. The Transformation

In the following, a transformer is described that adds fault-containment to any given silent protocol P that is self-stabilizing under the central daemon scheduler. The obtained protocol will be called P_F and is self-stabilizing with respect to a new predicate $Legit_{P_F}$, again under the central daemon scheduler. But protocol P_F also recovers from single transient faults in constant time.

The basic idea is captured in Figure 1. Every node executes cycles of the state machine shown there. Figure 1 also names certain actions that are to be performed along the different transitions from one state to another. Starting in state PAUSED, a node first tries to repair its primary state if necessary. It then executes a single move of protocol P. Then the node will ask all of its neighbors to copy its primary state - that is the so-called book-keeping. These copies serve as a backup. In the

case that the primary state of the node has been corrupted, they are used to restore the original value of the node's primary state. To execute several moves of protocol P, a node has to execute several cycles of the state machine.

Additional work is needed to use the state machine for fault-containment. The main objectives are summarized by the following three challenges. *Challenge 1:* After a fault, the state machine of each node should be guaranteed to start in the state PAUSED. Otherwise, a fault can essentially bypass the repair mechanism, for example by putting a node into the state EXECUTED. Then the book-keeping is performed before any repair takes place. This would destroy the information necessary to restore the nodes primary state. *Challenge 2:* After a fault, all neighbors of the faulty node must be kept from executing protocol P, at least until the primary state of the faulty node has been repaired. Otherwise, the fault would spread to the neighbors and beyond. This contamination process is hard to reverse. *Challenge 3:* To repair their primary state, nodes with only a single neighbor depend on information provided by their only neighbor. The variable in which the information is provided, might be corrupted by the fault. After a fault, it must be guaranteed that the value of that variable is re-computed.

These challenges have been addressed by establishing a constant *dialog* between a node and its neighbors. To perform a transition, each node has to execute the following steps:

1. Ask neighbors for approval of transition
2. Wait for acknowledgement of all neighbors
3. Perform transition

All neighbors execute the counterpart:

1. Wait for query
2. Respond with acknowledgement

This dialog turns out to be sufficient, to solve all of the three challenges given above. How this is achieved in detail is described in the next section.

4.1. Implementation

The implementation is based on the novel notion of cells. A *cell* v spans node v as well as all its neighbors. Node v is called *center node* and its neighbors are called *responding nodes*. Within cell v, the center node executes protocol Q whereas each responding node executes protocol R_v. Note that there is only one generic protocol R_v which is instantiated for different values of v.

A very important principle in the following approach is that every node of the system is a center node of a cell. Hence, cells overlap and two cells are called *neighboring cells*, if their center nodes are neighbors. In conclusion, every node of the system is not only a center node, but also a responding node of up to Δ neighboring cells. This leads to the fact that each node does not only have to execute protocol Q, but also one instance of protocol R_v for each cell that it is a responding node of. All these protocols are merged to the combining composition denoted by $Q \circ (\circ_{u \in N(v)} R_u)$. This composition constitutes protocol P_F, the result of the transformation.

Within cell v, protocols Q and R_v implement the state machine as shown in Figure 1 based on a dialog as previously explained. The implementation of the transition-actions has been split out into the two procedures $action_Q$ and $action_{R_v}$.

The center node, which is executing protocol Q, has the main control within cell v. It decides, whether a transition is to be performed and to which state the transition leads. To perform a transition, the center node first asks all responding nodes for approval. They respond with acknowledgements, and only if all responding nodes have acknowledged, then the transition can be completed by the center node.

Let v denote a center node. Protocol Q uses the variable $v.s \in \mathbb{Z}_4$ to store the numerical value of the current state according to Figure 1. The variable $v.q \in \mathbb{Z}_4$ also stores a numerical state value. If $v.q$ differs from $v.s$, then the pair $(v.s, v.q)$ is a *query* for a transition from state $v.s$ to state $v.q$. If $v.q$ equals $v.s$, then the pair is called a *pause*. Note that not all possible queries and pauses are valid. Any query for a transitions not shown in Figure 1 and any pause in a state other than PAUSED is considered invalid. Such invalid values indicate an erroneous state and basically lead to a reset of the cell (explained below). To approve a transition, each responding node $u \in N(v)$ of cell v has a variable $u.r_v$. It is simply set to $v.q$ to acknowledge a query by node v. All the r_v variables of a cell are also referred to as the *response variables*. The variables $v.s$ and $v.q$ as well as all the response variables are called *dialog-variables*.

At the beginning of a cycle of the state machine, all dialog-variables of a cell equal PAUSED. The cell is then called *dialog-paused*. The center node v can start a new cycle of the state machine by setting the variable $v.q$ to REPAIRED. While the center node waits, all responding nodes acknowledge the query by setting their response variable to $v.q$. If that has happened, the cell is called *dialog-acknowledged*. The center node can then complete the transition by setting $v.s$ to $v.q$. If a pause is not valid in the current state, then the center node asks for a new transition straight away by incrementing $v.q$. The last steps are repeated, until a cycle of the state machine is completed and the cell is dialog-paused again. During the whole process, a cell always stays within a state, that is formalized by notion of a *dialog-consistent* cell: The variables $v.s$ and $v.q$ form a valid pause or query and the response variables equal either $v.s$ or $v.q$. The following predicates reflect the above:

$$\begin{aligned} \textit{validQuery}(v) &\equiv v.q = (v.s + 1) \bmod 4 \\ \textit{dialogConsistent}(v) &\equiv (v.q = v.s = \text{PAUSED} \lor \\ &\qquad \textit{validQuery}(v)) \land \\ &\qquad \forall u \in N(v) : u.r_v \in \{v.s, v.q\} \\ \textit{dialogAcknowledged}(v) &\equiv \textit{validQuery}(v) \land \\ &\qquad \forall u \in N(v) : u.r_v = v.q \\ \textit{dialogPaused}(v) &\equiv v.q = v.s = \text{PAUSED} \land \\ &\qquad \forall u \in N(v) : u.r_v = \text{PAUSED} \end{aligned}$$

Note that *dialogAcknowledged*(v) and *dialogPaused*(v) are

Protocol P_F

Data: v is the current node

$G_Q(v)$ **or** $\exists u \in N(v) : G_{R_u}(v) \longrightarrow$ [RULE 1]
- **if** $G_Q(v)$ **then**
  - execute S_Q
- **foreach** $u \in N(v)$ **do**
  - **if** $G_{R_u}(v)$ **then**
    - execute S_{R_u}

Protocol Q

Data: v is the current node

($v.s \neq$ PAUSED **or** $v.q \neq$ PAUSED)
and not *dialogConsistent*$(v) \longrightarrow$ [RULE 1]
- $v.s :=$ PAUSED
- $v.q :=$ PAUSED

dialogPaused(v)
and *startCond*$_Q(v) \longrightarrow$ [RULE 2]
- $v.q :=$ REPAIRED

dialogAcknowledged$(v) \longrightarrow$ [RULE 3]
- *action*$_Q(v)$
- $v.s := v.q$
- **if** $v.s \neq$ PAUSED **then**
  - $v.q := (v.s + 1) \bmod 4$

Protocol R_v

Data: u is the current node
Data: v is the center node

$v.s = v.q =$ PAUSED
and $u.r_v \neq$ PAUSED $\longrightarrow$ [RULE 1]
- $u.r_v :=$ PAUSED

validQuery(v) **and** $u.r_v = v.s$
and not *waitCond*$_{R_v}(u) \longrightarrow$ [RULE 2]
- *action*$_{R_v}(u)$
- $u.r_v := v.q$

mutually exclusive and both imply dialog-consistency.

Of course, in the initial state or in case of a fault, cells are in general not dialog-consistent. To restore their dialog-consistency, the cells perform a reset in two steps: First, the center node executes Rule 1 of protocol Q. The responding nodes follow, and execute Rule 1 of protocol R_v. After the reset, all dialog-variables equal PAUSED. The cell is now dialog-paused and thus dialog-consistent.

For the different transition-actions shown in Figure 1, protocol Q maintains an additional variable $v.p \in \sigma_P$. The variable $v.p$ is the only primary variable of cell v. Hence, it is also referred to as the *primary state* of the cell. Furthermore, each responding node $u \in N(v)$ of cell v has a variable $u.d_v \in \mathbb{Z}_3$ and a variable $u.c_v \in \sigma_P$. The set σ_P denotes the set of local states with respect to protocol P. All the variables d_v within cell v are called *decision-variables* and the variables c_v are called *copy-variables*.

During the executing of a cycle of the state machine, the cell first performs the transition PAUSED → REPAIRED. Along with the acknowledgements, every responding node sets its decision-variable. Then the center node completes the transition by repairing its primary state. How exactly the repair mechanism works and how the decision-variables are used, is described in detail in Section 4.2. During the next transition to state EXECUTED, the center node will execute a single move of protocol P, simply by invoking S_P if G_P is true. Without loss of generality, G_P and S_P are assumed to directly access the variable $v.p$ on each node v. The transition to the state COPIED completes the cycle. During this transition, the responding nodes perform the book-keeping by updating their copy-variables along with acknowledging the query by the center node.

After completing a cycle of the state machine, all copy-variables equal the primary state of the cell. In this state, the cell is called *copy-consistent*. But during stabilization, the copy-consistency of a cell is destroyed regularly. For example by executing a move of protocol P. It changes the primary state of the cell, but not the copy-variables. For this and other reasons, copy-consistency cannot serve as an indicator, whether a fault is already repaired or not. A new predicate *repaired*(v) is defined:

$$\begin{aligned} \textit{copyConsistent}(v) &\equiv \forall u \in N(v) : u.c_v = v.p \\ \textit{repaired}(v) &\equiv \textit{copyConsistent}(v) \lor \\ &(\textit{dialogConsistent}(v) \land \\ &(v.s = \text{REPAIRED} \lor v.s = \text{EXECUTED}) \land \\ &\forall u \in N(v) : (u.r_v = \text{COPIED} \Rightarrow u.c_v = v.p)) \end{aligned}$$

A cell is called *repaired*, if it is copy-consistent or if it is dialog-consistent and in the state REPAIRED or EXECUTED. In addition, every responding node that has acknowledged the transition to COPIED, must have updated its copy-variable.

To keep a cell from completing a transition, a responding node can delay its acknowledgement. In the implementation, this controlled by the Boolean predicate *waitCond*$_{R_v}$. A responding node does not acknowledge the query of the center node, as long as the predicate is true. If such a responding node exists within a dialog-consistent cell, then the cell is called *blocked*.

In the current implementation, this mechanism is only used to keep a cell from finishing the transition REPAIRED → EXECUTED. Blocking a cell during this particular transition prevents the center node from executing a move of protocol P. To serve this special purpose, the wait-condition is defined as follows:

$$\textit{waitCond}_{R_v}(u) \equiv v.q = \text{EXECUTED} \land \neg \textit{repaired}(u)$$

Procedure $action_Q(v)$

Data: v is the current node

```
1  if v.q = REPAIRED
   and not copyConsistent(v) then
2      u := any neighbor of node v
3      if |N(v)| > 1 and ∀w ∈ N(v) : w.c_v = u.c_v then
4          v.p := u.c_v
5      if |N(v)| = 1 and u.d_v = UPDATE then
6          v.p := u.c_v
7      if |N(v)| = 1 and u.d_v = SINGLE and v.c_u = u.p
       and ( G_P(v : v.p, u : u.p) or G_P(u : u.p, v : v.p) )
       then
8          v.p := u.c_v
9      if |N(v)| = 1 and u.d_v = SINGLE and v.c_u ≠ u.p
       and not ( G_P(v : u.c_v, u : u.p) or
       G_P(u : u.p, v : u.c_v) ) then
10         v.p := u.c_v

11 if v.q = EXECUTED and G_P(v) then
12     execute S_P
```

Later on, the predicate *repaired*(v) is shown to be an invariant. That is, once a cell is repaired, it remains repaired under the execution of the state machine. During stabilization, blocked cells only exist in an initial phase. Once all cells are repaired, all cells are completely independent. No cell can become blocked.

During stabilization of protocol P_F, cells will frequently become dialog-paused. A cell only starts a new cycle of the state machine in certain situations. Certainly, cell v should start a new cycle, if it is not repaired. By executing a cycle of the state machine, the cell becomes repaired again. Second, cell v should start a new cycle, if protocol P is enabled for the center node, so that node v is given the chance to execute a move of protocol P. The start-condition is defined as follows:

$$\textit{startCond}_Q(v) \equiv \neg \textit{copyConsistent}(v) \ \vee \ G_P(v)$$

Note that the start-condition is only evaluated, if the cell is dialog-paused. In this case a test for *repaired*(v) can be replaced by a test for *copyConsistent*(v). The two predicates are equivalent for dialog-paused cells.

An important aspect of the implementation is the avoidance of deadlocks. It seems likely that cells can form a circle, in which each cell is blocked by its predecessor within the circle. From the definition of $\textit{waitCond}_{R_v}$ it follows that a cell can only be blocked if it is in the state REPAIRED. But by definition, such a cell is repaired. In other words: a blocked cell cannot block any other cell. Hence this kind of deadlock is impossible.

Procedure $action_{R_v}(u)$

Data: u is the current node
Data: v is the center node

```
1  if v.q = REPAIRED then
2      if |N(u)| = 1 then
3          u.d_v := SINGLE
4      else if |N(u)| > 1
       and ( ∀w ∈ N(u) : w = v ∨ w.c_u = u.p )
       and ( G_P(u) or G_P(v : v.p, u : u.p) ) then
5          u.d_v := UPDATE
6      else
7          u.d_v := KEEP
8  if v.q = COPIED then
9      u.c_v := v.p
```

4.2. Fault-Repair

In Section 5.1, protocol P_F is shown to converge to the set of legitimate states induced by $\textit{Legit}_{P_F}$. In such a legitimate state, all cells are legitimate. A cell is called *legitimate*, if it is dialog-paused, copy-consistent, and if protocol P is disabled for the center node.

$$\textit{Legit}_{P_F} \equiv \forall v \in V : (\neg G_P(v) \ \wedge \ \textit{dialogPaused}(v) \ \wedge \ \textit{copyConsistent}(v))$$

A 1-faulty state differs from a legitimate state in the variables of single node only. Something very similar holds for an individual cell. In a 1-faulty state, a cell is either legitimate, or it differs from a legitimate cell in the variables of a single node only. Notice that there can only be up to $\Delta + 1$ non-legitimate cells in a 1-faulty state.

Let the global state be 1-faulty and let v denote a cell that is not legitimate. If the fault has corrupted dialog-variables, then cell v differs from a dialog-paused cell in the dialog-variables of a single node. If cell v is not dialog-consistent, then it will be dialog-paused again after at most one move of either protocol Q or R_v. Note that this move does not change any of the copy-variables or the primary state of the cell. It may also happen that the cell is still dialog-consistent even though dialog-variables have been corrupted. This can only be the case, if only one of the two variables $v.s$ or $v.q$ was corrupted by the fault. Either $v.s$ was set to COPIED or $v.q$ was set to REPAIRED.

In any case it holds that the first transition-actions executed by a cell are that of the transition PAUSED → REPAIRED. First, all responding nodes of cell execute $action_{R_v}$ and set their decision-variables. Then the center node executes $action_Q$ and completes the repair. The repair mechanism is inspired by the synchronous protocol C described in [9]. The details of the repair mechanism are explained from the point of view of cell v. The explanation is split up into three special cases.

4.2.1. Multiple copy-variables. If there are multiple responding nodes in cell v, then it is easy to decide, whether the primary state has been corrupted. Prior to the fault, cell v must have been copy-consistent. If the global state is 1-faulty and if cell v is not copy-consistent, then cell v must match one of the following cases:

Case a) The variable $v.p$ differs from the copy-variables. All copy-variables have an identical value.

Case b) There is one copy-variable that differs from $v.p$. All other copy-variables have the same value as $v.p$.

The first case can be identified simply by checking, whether all copy-variables have the same value. If this is the case, then $v.p$ must be the variable which has been corrupted. Otherwise, it can be assumed that a copy-variable has been corrupted by the fault. Exactly this check is implemented in line 3 of procedure $action_Q$. If case a) is detected, then the value of $v.p$ is overridden with the value of one of the copy-variables. Note that a cell with only one responding node trivially matches both cases.

4.2.2. Only a single copy-variable. Let cell v be a cell with only one responding node and let u denote the cell neighboring to cell v. It is assumed that cell u has multiple responding nodes. Otherwise, the system consists of two nodes only. This case is discussed in Section 4.2.3.

If cell v is not copy-consistent, then either $v.p$ or $u.c_v$ has been corrupted by the fault. Since $u.c_v$ is the only copy-variable, it is not possible for node v to find out by itself which of the two variables it is. To solve the problem, node v lets node u decide. Node u can evaluate the guard of protocol P for node v, since u is the only neighbor of v. If protocol P is enabled for one of the two nodes, then either $u.p$ or $v.p$ must have been corrupted by the fault. Additional checks are performed by node u to make sure that it is $v.p$ and not $u.p$ which has been corrupted.

Case a) Assume that only $v.p$ and possibly $v.c_u$ have been corrupted by the fault. Then cell u must be copy-consistent, with the exception of $v.c_u$. Then the condition in line 4 of $\mathit{action}_{R_v}(u)$ is true depending on whether one of the two nodes is enabled. The decision-variable $u.d_v$ is set to UPDATE accordingly.

Case b) Assume that only $u.c_v$ and possibly $u.p$ have been corrupted by the fault. If $u.p$ has been corrupted, $u.p$ differs from all copy-variables in cell u. Hence, the condition in line 4 of $\mathit{action}_{R_v}(u)$ is false. If $u.p$ has not been corrupted, then cell u is copy-consistent, but protocol P is disabled for both the two nodes v and u. Again, the condition in line 4 of $\mathit{action}_{R_v}(u)$ is false. Node u set its decision-variable to KEEP and cell v does not change its primary state.

Cell u can also finish the transition PAUSED $\rightarrow$ REPAIRED prior to cell v. If $u.p$ has been corrupted, then its original value is restored, and protocol P is disabled on both nodes. In this case, the decision-variable $u.d_v$ is again set to KEEP and cell v does not change its primary state.

To avoid an additional handshake between nodes v and u, node u always assumes that it is the only neighbor of node v. In line 4 of procedure $\mathit{action}_{R_v}(u)$, node u evaluates the guard of node v in a virtual topology where u is the only neighbor of v. The result of this evaluation only affects the decision-variable $u.d_v$. Node v then decides, whether the assumption made by node u is correct: The procedure $\mathit{action}_Q(v)$ only uses the value of $u.d_v$, if it holds $|N(v)| = 1$.

4.2.3. The single-edge case. Let the system consist of nodes v and u only. When performing the transition PAUSED $\rightarrow$ REPAIRED, both nodes set their decision-variables to SINGLE. In this case, only lines 7 and 9 of procedure $action_Q$ apply. Note that it is possible for both nodes to evaluate guards for each other.

Let cell u be copy-consistent and let cell v be not copy-consistent. Then either $v.p$ or $u.c_v$ has been corrupted by the fault. By line 7 of procedure $action_Q$, cell v only updates its primary state, if one of the two nodes is currently enabled. In this case, it must have been $v.p$ that has been corrupted by the fault and setting $v.p$ to $u.c_v$ disables protocol P for both nodes.

Now, assume that both cells are not copy-consistent. In this case, either overriding $v.p$ with $u.c_v$ or overriding $u.p$ with $v.c_u$ must disable protocol P for both nodes. Which primary state is to be overridden is tested in line 9 of procedure $action_Q$. Without loss of generality, it is assumed that the check succeeds for node v and that node u has either not executed procedure $action_Q$ yet or that check has failed for node u. Node v then performs the assignment $v.p := u.c_v$. Cell v is now copy-consistent and protocol P is disabled on both nodes. If node u executes procedure $action_Q$ after node v, then cell v is already copy-consistent. But the condition in line 7 of procedure $action_Q$ is false because protocol P is disabled for both nodes and hence node u does not change its primary state.

5. Analysis

First, it is shown that protocol P_F is self-stabilizing with respect to Legit_{P_F}. Second, it is shown that protocol P_F is fault-containing with respect to Legit_P. In addition, an analysis of the fault-gap and the stabilization time is given.

The proofs are based on observations about the behaviour of individual cells. The behaviour of a cell is determined by protocols Q and R_v. But since protocol P_F is only a composition of these protocols, all the observations also apply to protocol P_F itself. Note, if a node is enabled with respect to protocol Q or any of the instances of protocol R_v, then it is also enabled with respect to protocol P_F. And if a node executes a move of protocol P_F, then it always executes a move of protocol Q or one of the instances of protocol R_v. Actually, counting the moves of protocols Q and R_v instead of the moves with respect to protocol P_F can only lead to an overestimation.

In the following, a cell is called *enabled*, if the center node is enabled with respect to protocol Q or if one of the responding nodes is enabled with respect to protocol R_v. It is said that

cell v makes a *move*, if the center node v makes a move with respect to protocol Q or if one of the responding nodes makes a move with respect to protocol R_v.

5.1. Proof of Self-Stabilization

Lemma 1. *If a cell is not dialog-consistent, then it is enabled and becomes dialog-consistent within the next 2 rounds and in at most 2Δ moves.*

Proof: Let v denote a cell that is not dialog-consistent. First round: If $(v.s, v.q)$ is a valid query, then the response variable of at least one responding node is not set correctly. The other up to $\Delta - 1$ responding nodes may execute Rule 2 of protocol R_v. If $v.s$ and $v.q$ don't equal PAUSED yet, then the center node now performs a reset by executing Rule 1 of protocol Q. Second round: Up to Δ responding nodes execute Rule 1 of protocol R_v. Cell v is now dialog-paused and hence dialog-consistent. □

Lemma 2. *A dialog-consistent cell v remains dialog-consistent under the execution of Q and R_v.*

Proof: Let v denote a dialog-consistent cell. Rule 1 of protocol Q as well as Rule 1 of protocol R_v cannot be enabled. All other rules set the dialog-variables in a way that complies with the definition of dialog-consistency. □

Lemma 3. *If a cell is not repaired, then it is enabled and becomes repaired within the next 7 rounds and in at most $3\Delta + 3$ moves.*

Proof: Let v denote a cell that is not repaired. If cell v is initially not dialog-consistent, then it becomes dialog-paused by Lemma 1. Then, if not yet copy-consistent, cell v executes the transition PAUSED → REPAIRED. If cell v is initially dialog-consistent, then the worst case is that cell v has to complete the transition EXECUTED → COPIED and execute the transitions COPIED → PAUSED → REPAIRED to become repaired.

Completing the transition EXECUTED → COPIED takes at most 2 rounds and Δ moves and the transition COPIED → PAUSED takes at most 2 rounds and $\Delta + 1$ moves. The transition PAUSED → REPAIRED takes 3 rounds and $\Delta + 2$ moves because of one extra move for executing Rule 2 of protocol Q. Note that $startCond_Q(v)$ is true if cell v is dialog-paused but not repaired. □

Lemma 4. *A repaired cell v remains repaired under the execution of Q and R_v.*

Proof: Let v denote a repaired cell. Rule 2 of protocol R_v sets the copy-variables to the the value of $v.p$. This cannot reduce the copy-consistency of cell v. It rather restores copy-consistency of cell v. Rule 3 of protocol Q can destroy copy-consistency by changing the primary state of the cell. But this can only happen, if cell v is dialog-consistent and is completing a transition to REPAIRED or EXECUTED. After that move, cell v is still dialog-consistent by Lemma 2. □

Theorem 5. *If $Legit_{P_F}$ is false, then there is at least one node enabled with respect to protocol P_F.*

Proof: Let v denote a cell that is not legitimate. Cell v is either not copy-consistent, not dialog-paused, or protocol P is enabled for node v. If cell v is not dialog-paused and blocked, then there is a neighboring cell that is not repaired and hence enabled by Lemma 3. If cell v is not dialog-paused and not blocked, then it is either enabled by Lemma 1 or in the middle of completing a cycle of the state machine and thus enabled. If cell v is dialog-paused, then it is either not copy-consistent or protocol P is enabled for node v. Then $startCond_Q(v)$ is true and hence Rule 2 of protocol Q is enabled. □

Theorem 6. *If $Legit_{P_F}$ is true, then all nodes are disabled with respect to protocol P_F.*

Proof: Assume that $Legit_{P_F}$ is true and let v denote a legitimate cell. Then cell v is dialog-paused and hence protocol R_v is disabled for all responding nodes. Rules 1 and 3 of protocol Q are disabled as well. Rule 2 of protocol Q depends on $startCond_Q(v)$ which is false, since cell v is copy-consistent and protocol P is disabled for node v. □

Theorem 7. *Protocol P_F terminates after a finite number of moves.*

Proof: Modifications of the primary state of a cell only result from either the execution of protocol P or from a direct modification during the repair of a cell. As long as there are no direct modifications, the number of possible moves of protocol P is finite since protocol P is silent. After each direct modification by a repair, the stabilization process of protocol P starts over. This happens at most n times, since each cell is repaired at most once by Lemma 4. In conclusion, the total number of modifications of primary states is finite.

Let v denote a cell. A complete cycle of the state machine takes at most $4\Delta + 5$ moves. By Lemma 3, cell v is repaired after a finite number of moves. After these moves, cell v only starts a new cycle, if protocol P is enabled on node v. By the time cell v is about to finish the transaction REPAIRED → EXECUTED, protocol P might be disabled on node v. Then at least one cell neighboring to v must have modified its primary state. Otherwise, $v.p$ is modified by the execution of protocol P. At least one modification of a primary state happends during each cycle of v. Hence, the number of cycles executed by cell v must be finite. □

Theorem 8. *Protocol P_F is silent and self-stabilizing.*

5.2. Proof of Fault-Containment

In the following, the initial state is assumed to be a 1-faulty state that has been derived from a legitimate state by perturbing variables of node v only. There may exist multiple choices of a legitimate state and a node v which yield the same 1-faulty state. The proofs hold for all of them.

Let $Legit_P$ be the predicate that decides whether the primary state is legitimate with respect to protocol P. Where possible,

the stricter predicate $Legit^*_P$ will be used.

$$Legit^*_P \equiv \forall v \in V : \neg G_P(v)$$

The predicate is true if and only if protocol P is disabled on all nodes. It is easy to see that $Legit_{P_F} \Rightarrow Legit^*_P$ holds. Since protocol P is silent, $Legit^*_P \Rightarrow Legit_P$ must also hold.

Note that in the initial state, all cells $w \notin N(v) \cup \{v\}$ are legitimate and therefore disabled. They stay legitimate and disabled, if none of the cells in $N(v)$ change their primary state. That is shown be the case.

Due to the page limitation, the proofs have been shortened and do not cover the case that the system consists of a single edge only. A discussion of this case is given in Section 4.2.3.

In a 1-faulty initial state, cells differ from dialog-paused cells in the dialog-variables of at most one node. In addition to the discussion at the beginning of Section 4.2, a modified version of Lemma 1 applies. It shows that cells that are not already dialog-consistent, perform a reset and become dialog-paused within the first round and without changing their primary state or copy-variables. Cells, of which only the dialog-variables have been corrupted, may become legitimate within the first two rounds. The first transition that all other cells execute, is PAUSED → REPAIRED. The proofs focus on this and the following transitions. Note that none of the acknowledgements of this particular transition can be part of the 1-faulty initial state. This shows that the value of the decision-variables is always re-computed after a fault. Otherwise, their value could induced by a fault which would lead to unwanted behaviour.

Lemma 9. *As long as cell v is not repaired, no cell neighboring to v changes its primary state or copy-variables.*

Proof: It can be assumed that $v.p$ has been corrupted and that cell has not yet completed the transition to state REPAIRED yet. Otherwise, cell v would be repaired. Now let u denote a cell neighboring to v.

If node u has multiple neighbors, then cell u does not perform a repair by either line 1 or line 3 of $action_Q(u)$, depending on whether $v.c_u$ has been corrupted.

If node u has only one neighbor, then the condition in line 4 of $action_{R_u}(v)$ is false because $v.p$ has been corrupted. Hence, $v.d_u$ is set to KEEP.

In all cases, cell u becomes blocked if it attempts to execute the transition REPAIRED → EXECUTED. □

Lemma 10. *Let c be the first global state, in which cell v is repaired. Then $Legit^*_P(c)$ holds.*

Proof: If $v.p$ has not been corrupted, then c is the initial state and $Legit^*_P(c)$ holds. If $v.p$ has been corrupted, then cell v must complete the transition to state REPAIRED to become repaired. If node v has multiple neighbors, then $action_Q(v)$ will restore the value of the primary state in line 4. That move yields c. Now, assume that node v has only one neighbor and let u denote this neighbor. Cell u is either copy-consistent, with the exception of $v.c_u$. It depends on the evaluation of the guards, whether the condition in line 4 of $action_{R_v}(u)$ is true. Hence, $u.d_v$ is set to UPDATE only if protocol P is enabled for node u or v. Otherwise, cell v does not need to change its primary state and $u.d_v$ is set to KEEP. Depending on $u.d_v$, the primary state of cell v is set to $u.c_v$ in line 6 of $action_Q(v)$. That move yields c. □

Theorem 11. *Let c the be first global state for which $Legit^*_P$ holds. Then $Legit^*_P$ holds for all subsequent global states.*

Proof: Let $\langle c_0, c_1, c_2, \ldots \rangle$ be an execution of protocol P_F. Let c_i be the first global state, in which cell v is repaired. The move that yields c_i will be first that alters the primary state of cell v. By Lemma 9, no cell neighboring to v changes its primary state prior to c_i. Hence, if $Legit^*_P$ holds for any global state $c_k, k < i$, then it holds for all $c_j, j = k, \ldots, i-1$ and $Legit^*_P(c_i)$ holds by Lemma 10.

Starting in c_i, no cell will change its primary state by execution of protocol P as long as $Legit^*_P$ holds. It remains to show that no cells changes its primary state by repair. For a cell u neighboring to v, this follows from either lines 1 and 3 of $action_Q(u)$ or line 4 of $action_{R_u}(v)$. If cell v is already repaired before reaching state REPAIRED, then it has to be copy-consistent and it won't perform any repair because of line 1 of $action_Q(v)$. □

Corollary 12. *Let c the be first global state for which $Legit_P$ holds. Then $Legit_P$ holds for all subsequent global states.*

Theorem 13. *The worst-case containment time of protocol P_F is 4 rounds.*

Proof: Since the initial state is 1-faulty, a modified version of Lemma 3 shows that cell v becomes repaired within the first 4 rounds. Then Lemma 10 applies. □

Theorem 14. *The worst-case contamination number of protocol P_F is 1.*

Proof: Follows directly from Lemmas 9 and 10. □

Theorem 15. *Protocol P_F is fault-containing with respect to $Legit_P$.*

5.3. Fault-Gap and Stabilization Time

Theorem 16. *The worst-case fault-gap is* 10 *rounds.*

Due to the page restriction, we only give a short sketch of the proof: Since the initial state is 1-faulty, a modified version of Lemma 3 shows that each cell reaches state REPAIRED within the first 4 rounds. Then none of the cells is blocked and they complete the current cycle of the state machine by the end of round 10. At the end of the cycle, copy-consistency has been restored and by Lemma 10 and Theorem 11 protocol P is disabled for all center nodes at the end of the cycle. So all cells are legitimate.

Lemma 17. *Let $\langle c_0, c_1, c_2, \ldots \rangle$ be an execution of P_F and let c_i denote the first state of the execution, in which all cells are dialog-consistent and repaired. If protocol P stabilizes from c_i in at most T rounds, then protocol P_F stabilizes from c_0 in at most $9T + 15$ rounds.*

Again, due to the page restriction, we only give a short sketch of the proof: Observe that a dialog-consistent and repaired cell does not change its primary state by any action other than executing protocol P. To execute one round of protocol P, protocol P_F has to give each node the chance to execute a move of P. It can be shown that this is the case within nine rounds of protocol P_F, simply because a complete cycle of the state machine takes nine rounds. There are two exceptions: First, a cell could be blocked. But this cannot be the case, since it is assumed that all cells are repaired. Second, the start-condition $startCond_Q$ could be false for a dialog-paused cell. A dialog-paused repaired cell is always copy-consistent. Hence, the start-condition can only be false if protocol P is disabled for the center node. But then the current round of protocol P is already finished for that node.

The 15 rounds in addition to the $9T$ rounds consist of the seven rounds in which all cells become dialog-consistent and repaired (See Lemmas 1 and 3) and another eight rounds for all cells that might have to complete a just begun cycle of the state machine after protocol P has finished stabilizing. Leaving such minor terms aside, this leads to the following observation about the worst-case stabilization time:

Theorem 18. *The slowdown-factor of protocol P_F over protocol P is* 9.

6. Concluding Remarks

Asymptotically, the worst-case stabilization time of protocol P_F differs from that of protocol P by a constant slow-down factor only. Compared to protocol P, the space required by protocol P_F increases by a factor of about $\Delta + 1$. Methods on how to lower the slow-down and space requirements are currently investigated. An alternative repair method [9] that causes only temporary space-overhead is not applicable.

In contrast to the transformation in [9], protocol P_F does not depend on any knowledge about the size of the system. Hence, it scales well to systems of any size at run-time. A further advantage is that faults are tightly contained, even concerning their impact on secondary variables. In fact, only cells neighboring to the faulty node become enabled.

The fault-gap of protocol P_F is constant. It does not depend on the size of the system or the degree of the nodes. This is an improvement over all known general transformations for asynchronous system. It makes the transformation suitable for systems with a large number of nodes and greatly increases their availability.

Since the repair mechanism is based on local knowledge only, protocol P_F always successfully repairs multiple faults, if the minimal distance between the faulty nodes is large enough. A minimal fault-distance of 4 should be sufficient. For systems with high average node degree, the repair mechanism can be modified to use a majority vote to determine the correct primary state. Then protocol P_F may even recover from multiple faults within the same neighborhood.

Protocol P_F does not explicitly rely on unique node-identifiers. But every node executes multiple instance of protocol R_v, each one designated to a different center node. The center nodes must be able to identify the variables designed to them. For example, locally unique identifiers provide a suitable mechanism. Another suitable mechanism is port-numbering [1], but the definition does not have a natural equivalent in the shared memory model.

References

[1] D. Angluin, "Local and global properties in networks of processors (extended abstract)," in *Proceedings of the 12th annual ACM Symposium on Theory of Computing*. ACM, 1980, pp. 82–93.

[2] J. Beauquier, S. Delaët, and S. Haddad, "A 1-strong self-stabilizing transformer," in *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2006, pp. 95–109.

[3] J. Burman, T. Herman, S. Kutten, and B. Patt-Shamir, "Asynchronous and fully self-stabilizing time-adaptive majority consensus," in *9th International Conference on Principles of Distributed Systems*. Springer, 2006, pp. 146–160.

[4] A. Dasgupta, S. Ghosh, and X. Xiao, "Probabilistic fault-containment," in *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2007, pp. 189–203.

[5] ——, "Fault-containment in weakly-stabilizing systems," in *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2009, pp. 209–223.

[6] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[7] S. Ghosh and A. Gupta, "An exercise in fault-containment: Self-stabilizing leader election," *Information Processing Letters*, vol. 59, no. 5, pp. 281–288, 1996.

[8] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju, "Fault-containing self-stabilizing algorithms," in *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 45–54.

[9] ——, "Fault-containing self-stabilizing distributed protocols," *Distributed Computing*, vol. 20, no. 1, pp. 53–73, 2007.

[10] S. Ghosh, A. Gupta, and S. V. Pemmaraju, "A fault-containing self-stabilizing spanning tree algorithm," *Journal of Computing and Information*, vol. 2, no. 1, p. 322–338, 1196.

[11] ——, "Fault-containing network protocols," in *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, 1997, pp. 431–437.

[12] S. Ghosh and X. He, "Fault-containing self-stabilization using priority scheduling," *Information Processing Letters*, vol. 73, no. 3-4, pp. 145–151, 2000.

[13] T. Herman and S. Pemmaraju, "Error-detecting codes and fault-containing self-stabilization," *Information Processing Letters*, vol. 73, no. 1-2, pp. 41–46, 2000.

[14] T. C. Huang, "An efficient fault-containing self-stabilizing algorithm for the shortest path problem," *Distributed Computing*, vol. 19, no. 2, pp. 149–161, 2006.

[15] S. Kutten and B. Patt-Shamir, "Time-adaptive self stabilization," in *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*. ACM, 1997, pp. 149–158.

[16] L. Lamport, "Solved problems, unsolved problems and non-problems in concurrency," in *Proceedings of the 3rd annual ACM symposium on Principles of distributed computing*. ACM, 1984, pp. 1–11.

[17] J.-C. Lin and T. C. Huang, "An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 742–754, 2003.

[18] Y. Yamauchi, S. Kamei, F. Ooshita, Y. Katayama, H. Kakugawa, and T. Masuzawa, "Composition of fault-containing protocols based on recovery waiting fault-containing composition framework," in *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2006, pp. 516–532.

[19] ——, "Timer-based composition of fault-containing self-stabilizing protocols," in *Proceedings of the 2nd International Symposium on Intelligent Distributed Computing*. Springer, 2008, pp. 217–226.