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

Sven Köhler · Volker Turau

1 Introduction

The notion of self-stabilizing was coined by Dijkstra [6]. A self-stabilizing distributed system provably converges to a set of legitimate configurations in finite time, without any external intervention and regardless of its initial configuration. The system remains legitimate until a fault occurs. The work of Dijkstra was then picked up by Lamport [19] who called it a “milestone in work on fault-tolerance.” The system’s configuration after a transient fault can be regarded as simply another initial configuration. 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 configuration again. This is mostly due to the fact that each node of the system has only local knowledge often making it impossible to identify faulty information. The problem is worsened by the choices of the scheduler: it may chose nodes neighboring to the faulty node first before the faulty node itself can take any counter measures. The moves of selected nodes allow the faulty information to contaminate their own state, which in turn allows the information to spread to even more nodes. This process is called contamination.

Fault-containing algorithms promise quick recovery from small scale faults which are usually much more frequent than large scale faults. Handling them efficiently, that is preventing contamination, can greatly increase the availability of the system. The first work on the subject of fault-containing self-stabilization by Ghosh et al. [8,10] already established that fault-containment and self-stabilization are not contradictory goals. In fact, the work by Ghosh et al. showed that any silent self-stabilizing algorithm can be made fault-containing by establishing a certain form of redundancy. This makes it possible for a node to detect corruptions of its neighbor’s state and leads to a mechanism that effectively circumvents contamination.

There exist several intuitive metrics for measuring the fault-containment qualities of a self-stabilizing algorithm [10]. 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 system’s output is 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. Another metric is the fault-gap. It indicates how frequently small scale faults may occur. A fault-containing algorithm is only guaranteed to handle two subsequent faults efficiently, if the time between them is not smaller than the fault-gap. The fault-gap is usually larger than the containment time. The algorithm may have to prepare for the containment of another fault after the output variables have been restored.

The main contribution of this paper is a new transformation that maps a silent self-stabilizing algorithm $A$ to a fault-containing self-stabilizing algorithm $A_C$. The containment time, contamination number, and also the fault-gap of $A_C$ are constant. The fault-gap is significantly lower than that of all previously known general transformations for asynchronous systems. In addition, the transformation is shown to preserve the stabilization time of $A$, except for a constant slow-down factor. The following Theorem summarizes the contribution of this paper:

**Theorem 1** (Main theorem) Let $A$ denote a silent self-stabilizing algorithm under the distributed scheduler (resp. the central scheduler). The transformed algorithm $A_C$ is a fault-containing extension of $A$ and is self-stabilizing under the distributed scheduler (resp. the central scheduler). The fault-gap and slow-down factor of $A_C$ are constant.

### 2 Model of computation

#### 2.1 Standard model

A distributed system is represented as an undirected graph $(V, E)$ where $V$ is the set of nodes and $E \subseteq V \times V$ is the set of edges. Let $n = |V|$ and $\Delta$ denote the maximal degree of the graph. 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 $N(v) \subseteq V$ and $N[v] = N(v) \cup \{v\}$. Each node stores a set of variables. The values of all variables constitute the local state of a node. Let $\sigma$ denote the set of possible local states of a node. The configuration of the system is the tuple of all local states in the system and $\Sigma = \sigma^n$ denotes the set of global states.

Nodes communicate via locally shared memory, that is every node can read the variables of all its neighbors. Write access to any but a node’s own variables is prohibited. Each node $v \in V$ executes a protocol consisting of a list of rules, each of the form $guard \rightarrow statement$. The guard is a Boolean expression over the variables of node $v$ and its neighbors. The statement consists of a series of commands. 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 makes a move by executing the statement of the enabled rule. A step is finished, if all nodes in $S_i$ have made their moves. Composite atomicity is assumed. This means each move is executed as an atomic action. The changes made by each individual move don’t become visible to the neighboring nodes before the end of the step. We distinguish between the central scheduler which selects exactly one enabled node per step ($|S_i| = 1$) and the distributed scheduler which may select multiple enabled nodes in each step ($|S_i| \geq 1$). No assumptions on the fairness of the scheduler are made.

An execution $e = \langle c_0, c_1, c_2, \ldots \rangle$, $c_i \in \Sigma$ is a sequence of configurations, where $c_0$ is the initial configuration and $c_i$ is the configuration after the $i$th step. In other words, if the current configuration is $c_{i-1}$ and all nodes in $S_i$ make a move, then this yields $c_i$. The corresponding sequence $S = \langle S_1, S_2, S_3, \ldots \rangle$, $S_i \subseteq V$ is called schedule.

Time is measured in rounds. Roughly speaking, each round provides just enough time such that each node is given the chance to make a move, provided that the node was enabled at the beginning of the round and doesn’t become disabled by a move of a neighboring node. Formally, an execution $e$ and the corresponding schedule $S$ is partitioned into rounds as follows: The first round of $e$ is the prefix $\langle c_0, c_1, \ldots, c_j \rangle$ of minimal length such that $V \setminus D_0 \subseteq \bigcup_{i=1}^{j} D_i \cup S_i$ where $D_i \subseteq V$ denotes the set of nodes disabled in $c_i$. The second and all further rounds are derived recursively by applying the definition of the first round to the suffix $e' = \langle c_j, c_{j+1}, c_{j+2}, \ldots \rangle$ and $S' = \langle S_{j+1}, S_{j+2}, S_{j+3}, \ldots \rangle$.

#### 2.2 Multi-protocol model

The standard model as defined above is extended to a multi-protocol model. It is not more powerful than the standard model, but it simplifies the implementation and proofs of the transformation described in this paper. The multi-protocol model allows multiple protocols per node rather than just a single one. An algorithm $A$ denotes a set of protocols. Each node runs all protocols in $A$. Protocol $p$ running on node $v$ is called instance of $p$ and is formally denoted by $(v, p) \in \mathcal{M}$ where $\mathcal{M} = V \times A$ is the set of instances.

A node designates a separate set of variables to each of its instances. These constitute the local state of the instance. Each instance $(v, p) \in \mathcal{M}$ can read all variables of the nodes in $N[v]$, no matter which protocol they belong to. Write access to any other variables than the ones designated to $p$ by $v$ is prohibited. The instance $(v, p)$ is called enabled if a guard of $p$ evaluates to true. Node $v$ is called enabled, if