Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Graph Algorithms in Harsh Conditions

Bachelor's Thesis

André Sousa Anjos

soandre@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Diana Ghinea, Jakub Sliwinski
Prof. Dr. Roger Wattenhofer

August 24, 2021

Acknowledgements

First of all I want to thank Prof. Dr. Roger Wattenhofer for giving me the chance to write my Bachelor's Thesis in his department.

I thank my supervisors Diana Ghinea and Jakub Sliwinski for their help, good ideas and support while working on this thesis.

I thank my family too for their help and support during this whole time.

Abstract

In this thesis we look at two interesting graph problems, namely distributed single-source shortest path and distributed graph coloring. While most works focus on obtaining fast algorithms, resilience is essential in distributed computing. Single-source shortest path and $(\Delta + 1)$-graph coloring are already proven to be feasible under transient faults (self-stabilization). In this thesis, we study their resilience when byzantine faults may occur in addition to transient faults (strict-stabilization).

For the single source shortest path problem we prove that achieving strict-stabilization is impossible, we present a solution that is "almost strict-stabilizing". To ensure that the byzantine nodes do not fake paths we requires digital signatures.

For the graph coloring problem we present three different solutions which all satisfy the definition of strict-stabilization. One solution for a central and fair daemon and two solution for a distributed daemon, a deterministic one and a more efficient randomized one.

Contents

Chapter 1

Introduction

Obtaining meaningful results even in the presence of faults is essential in distributed computing. An example of such failures is given by transient faults, where the state of any node in a system can be reset at any point in time. An algorithm that still obtains meaningful outputs under transient faults is called self-stabilizing, a notion introduced by Edsger W. Dijkstra in [1]. Other failures that may occur in distributed systems are byzantine faults, where nodes may deviate arbitrarily from the steps of the algorithm. Nesterenko and Aurora define in [2], and also in the paper [3], the notion of strict-stabilization: intuitively, an algorithm is strict-stabilizing if it obtains meaningful outputs in the presence of transient faults and of an unbounded number of byzantine nodes.

While classical graph problems are of great interest in distributed computing, most studies focus in obtaining close-to-optimal results quickly, and not resilience. For max matching [3] and edge-coloring [4], strict-stabilizing algorithms are known. We continue this line of work with two well-known problems: single-source shortest path and graph coloring.

For the single-source shortest path problem, we require the nodes to obtain a distance between the shortest distance in the original graph, and the shortest distance in the subgraph containing honest nodes only. We first show a negative result: there is no strict-stabilizing algorithm satisfying this definition. On the positive side, we present an algorithm that achieves a slightly weaker definition of strict-stabilization: instead of requiring honest (non-faulty) nodes to maintain their state after a solution is found unless a transient fault occurs, we allow them to change their state as long as the system still presents a solution to our single-source shortest path problem.

In the $(\Delta + 1)$-graph coloring we want to achieve a proper coloring so that no two neighbors have the same time. A negative point is that we cannot guarantee that nodes with byzantine neighbors will not change their state after reaching a solution. However, we show that it is possible for all other nodes without a byzantine neighbor. We present three different algorithms to solve this problem under different environments. One algorithm which works under a fair central daemon and two algorithms, a deterministic one and a randomized one, which

work under a distributed daemon.

1.1 Related work

There are many works which study self-stabilization or strict-stabilization for different problems. The work [1] of Edsger W. Dijkstra introduces self-stabilization. Dijkstra presents in this work an self-stabilizing algorithm to solve mutual exclusion. In the works [2] and [3] we find the definition for strict-stabilization which is stronger than self-stabilization because it is also resilient to byzantine behavior and not only transient faults. In [3] they present a strict-stabilizing solution for the maximal matching problem.

A good introduction to the single source shortest path problem can be found in [5]. The self-stabilizing solution presented uses shared memory between the nodes in a given graph, as we do in our solution, and weights for the edges.

The paper [6] gives a first idea how to solve the graph coloring problem. They give a self-stabilizing solution using a root node and a spanning tree of the given graph. A strict-stabilizing solution for edge-coloring can be found in [4]. They present a solution where neighbor nodes exchange lists with the colors of their edges. Then a node v can propose to a neighbor u a color based on the lists of v and u. Only if both nodes propose and agree on the same color c the edge between v and u gets color c.

Chapter 2

Strict-Stabilization

Self-stabilization is an exciting fault model in distributed systems. Since self-stabilization is not resilient to byzantine nodes, but our solutions should also work for nodes with byzantine behaviors, we decided to solve our problems for strict-stabilizing algorithms. We use the definition of strict-stabilization from [2] and [3]: Strict-stabilization combines self-stabilization, which is resilient to transient faults, with resilience to byzantine faults.

We use the definition of self-stabilization from [1]: A system is self-stabilizing if regardless of the initial state it is guaranteed to reach a legitimate state. If no more faults happen the system is guaranteed to remain in that legitimate state.

To be able to talk about self-stabilization we first need to define what is a state for each problem. A state is composed by the values of the variables of the honest nodes. Second we need to define the legitimate states. A legitimate state is a state where the variables need to satisfy some defined properties. When this is done we can start searching for an algorithm so that when starting from an arbitrary state the algorithm reaches a legitimate state in a finite amount of steps.

In a strict-stabilizing environment a state can be manipulated in two ways. One way are transient faults. These type of faults can change everything in a state, like change a variable to a new number or delete it completely. So when transient faults happen the old state changes automatically to a new state. In our problems we need to guarantee the self-stabilization after the transient faults stop happening. If they happen we can never guarantee the convergence to a legitimate state because they could change the state at any point in time. Beside honest nodes there are also byzantine nodes. These are corrupted nodes, meaning they do not follow the steps of the algorithm correctly and can have arbitrary behavior. For example they can lie to their neighbors or they can influence their neighbors to change their variables. With the changes they can trigger they can force the system to change to a new state. The algorithms we search must then guarantee that we reach a legitimate state of the problem and that we remain in this legitimate state if we reach one despite the presence of transient faults and byzantine nodes in order to satisfy the definition of strict-stabilization.

Chapter 3

Single-Source Shortest Path

In this section we are going to talk about a very interesting graph problem namely finding single-source shortest path in a given Graph G. G consists of different nodes. Every G has an honest source node, lets call it Alice from now on. Then there are honest nodes which all execute the same algorithm. Last there are byzantine nodes, these are corrupted nodes which may deviate arbitrarily from the steps of the algorithm. The goal is to find the shortest path from all honest nodes to the source node Alice and the algorithm should satisfy the definition of strict-stabilization.

3.1 Defining the problem

3.1.1 System

The system consists of a daemon, which is controlled by a central adversary, and a connected graph G with three types of nodes:

- Honest nodes

  All honest nodes execute the same algorithm. They follow the steps of the algorithm correctly. Their goal is to find their shortest path and their distance to Alice.

- Byzantine nodes

  Byzantine nodes, also called corrupted nodes, are controlled by a central adversary. They deviate from the protocol, they can act like an honest node or do something completely different and try to mislead the honest nodes. The corrupted nodes can always communicate with each other about everything.

- Source node called Alice

G contains one honest source node. Alice is always honest but has its own algorithm to execute. Alice is the source node because the other nodes in G want to find their shortest path to Alice.

The following holds for our system:

- Transient faults exist

  This means that every honest node can get its state reset at any time. A transient fault can erase or change the values of the variables the honest nodes need to store.

- Asynchronous model

  A node only makes a step if it gets activated.

- Asynchronous communication channels

  Every two adjacent nodes in G communicate through asynchronous communication channels: this means that any message is eventually delivered, but within an unknown amount of time.

The solution we present in this section works under a distributed daemon and under a fair central daemon.

We assume a public key infrastructure providing us with digital signatures for all the nodes in the graph. This means that all nodes have access to all public keys and each node has additionally access to its own secret key. $Sign_{sk}(P)$ is the signature for path P with secret key sk. For our proofs we assume that signatures are perfectly unforgeable and $\mathcal{A}$, the adversary, has access to the secret keys of the byzantine nodes. Further we assume that the keys of the honest nodes are not affected by transient faults.

Note that when replacing our ideal signatures with real signatures our results still hold with high probability, as long as the keys are changed regularly. This is because our algorithm is meant to run indefinitely, hence even an adversary with bounded computed power might be able to eventually forge the signatures.

3.1.2 Notations

We use the shared memory model to store the paths to Alice. Each node in the graph has the following variables.

- $P_{v,u}$

  To represent the shared memory between the nodes each node v stores a variable $P_{v,u}$ for each neighbor u. Node v stores in $P_{v,u}$ his current shortest

path to Alice. If a neighbor u wants to know the shortest path from v to Alice it only has to read the variable $P_{v,u}$. Initially $P_{v,u} =$ "$null$".

For example if v is a neighbor of Alice then it eventually sets

$P_{v,u} = \langle\langle HIv, sign_{Alice}(HIv)\rangle||HIu, sign_v(\langle HIv, sign_{Alice}(HIv)\rangle||HIu)\rangle$ for all neighbors u of v, where $||$ denotes the concatenation.

- P_N

  Each node v stores in P_N the current shortest path from the neighbors to Alice.

- d_v

  Every node v has a variable d_v to store the current shortest distance to Alice. Initially $d_v = \infty$.

We use $N(v)$ to denote the neighborhood of node v.

3.1.3 Conditions

Before we can tackle the problem and start searching for a strict-stabilizing algorithm we first need to define some conditions under which the algorithm has to perform and the properties the outputs should satisfy.

- Every two byzantine nodes are adjacent

  Since the byzantine nodes are controlled by the adversary they can communicate with each other. Therefore we can assume that they are connected with each other.

- Edges in G have cost 1

- $d_v = \infty$ if there is no path from an honest node v to Alice

- distance shortest path $\leq$ output distance $\leq$ distance of the shortest honest path

  For every honest node the output distance, which is the distance to Alice, must be in the bounds above. We can not guarantee that the actual shortest path is found because it could contain a corrupted node and if the corrupted node does nothing all the time we never find the actual shortest path. Since the output distance can never be smaller than the actual shortest distance the distance of the shortest path is the lower bound. As the byzantine nodes may simply crash, we set the upper bound to the distance of the shortest honest path, if there is one, else the nodes output ∞.

3.1.4 Legitimate state

Let H define the subgraph of G that does not contain any corrupted nodes. The legitimate state for this problem is:

- $\forall$ honest node v: $d_G(Alice, v) \leq d_v \leq d_H(Alice, v)$

  The current distance d_v between the honest node v and Alice has to be between the bounds above. It needs to be less or equal to $d_H(Alice, v)$, which denotes the distance of the shortest honest path in H. It needs also to be greater or equal to $d_G(Alice, v)$, which denotes the actual shortest path in G. If the distance would be smaller than the lower bound the distance would not be based on an actual path in G, we cannot have a better distance than the distance of the shortest path. If the distance is greater than the upper bound we can always find the smaller distance of the shortest honest path.

- $\forall$ honest node $v \in N(Alice)$: $d_v = 1$

  Every honest neighbor of Alice needs to have output distance 1. This is the case because every edge in G has cost 1 and all neighbors of Alice need only one edge to reach Alice.

- $d_{Alice} = 0$

  The distance from Alice to Alice needs to be 0.

3.2 Difficulties encountered

Before presenting the solution, we describe some issues that we have encountered.

We first considered the message passing model. Alice starts sending messages saying Hi to her neighbors. When an honest node receives a message it creates a new message by adding a new Hi to the old message. The honest node then sends the new message to its neighbors. To find the shortest distance to Alice the honest nodes count the number of Hi's in the messages they receive and store the smallest number. The problem of this idea is that the corrupted nodes can create fake messages of arbitrary distance and mislead the honest nodes.

We prevent such behavior by using digital signatures. Every honest node signs the new message with his private key before sending it to its neighbors. When receiving a message each honest node can then verify the message with the public key from the sender. This way they can be sure that the message is correct and it came from the right sender. The distance to Alice would be the number of signatures in the messages. Message signing prevents the corrupted nodes from creating fake messages because they cannot sign messages for honest nodes because we assume the signature scheme is unforgeable.

In the end our solution algorithm uses shared-memory between neighbors instead of sending the messages. It still uses message signing to prevent the corrupted nodes from creating arbitrary paths.

3.3 No strict-stabilizing algorithm possible

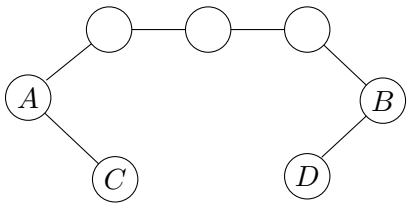
Before presenting our solution we first prove that achieving the standard strict-stabilizing definition for the single-source shortest path problem is actually impossible. In our proof, we consider three scenarios with three different graphs that are indistinguishable to an honest node named B.

Theorem 3.1. *There is no deterministic algorithm solving the strict stabilizing single-source shortest path problem.*

Proof. Assume there exists a strict-stabilizing deterministic algorithm Λ which solves the single-source shortest path problem. This means that, if no transient fault occurs, Λ reaches a legitimate state within a finite amount of time, and the system remains in that state.

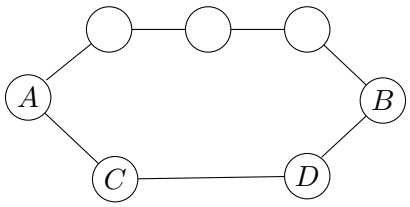
We consider the following scenarios:

- Scenario 1: Graph G_1



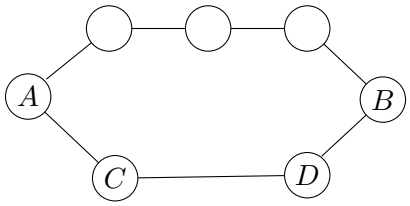
  All nodes are honest. Nodes C and D have transient faults.

- Scenario 2: Graph G_2



  All nodes are honest. Nodes C and D have transient faults such that, they behave identically to scenario 1.

- Scenario 3: Graph G_3



All nodes are honest except C and D, which are corrupted. Nodes C and D act as they have transient faults, they mimic C and D's behavior from scenario 1.

The daemon behaves the same in all three scenarios. B's view is identical in all three scenarios.

As Λ is strict-stabilizing, at some time k node B in scenario 1 will output distance 4 to node A. The graphs in scenarios 2 and 3 are indistinguishable to the honest nodes excepting C and D which act the same as nodes C and D from scenario 1. Hence the executions from scenarios 2 and 3 are identical to the execution from scenario 1. Therefore the time is the same and at time k node B in scenarios 2 and 3 also outputs distance 4 to node A and Λ never changes its outputs.

However, we obtain a contradiction. The shortest honest path from A to B in G_2 has length 3. Hence, by definition of our legitimate state B should output at most 3 in scenario 2.

□

3.4 Almost strict-stabilizing algorithm

In the previous section we proved that there is no strict-stabilizing algorithm. In this section we relax the definition of strict-stabilization by allowing the honest nodes to update their state as long as the system is still in a legitimate state even if they are already in a legitimate state, and we provide an algorithm satisfying this new relaxed definition. We call the relaxed definition: almost strict-stabilization.

The solution algorithm consists of two parts. One for the honest source node Alice and another for all the rest of the honest nodes in graph G.

3.4.1 Part of algorithm for Alice

First, Alice needs to update her shared memory with all neighbors. For this all the variables $P_{Alice,u}$ for all u in the neighborhood of Alice need to be updated. In $P_{Alice,u}$ Alice stores the current shortest path to herself. Second, Alice needs

to update the distance to herself. This is done by setting the variable d_{Alice} to 0.

Algorithm 1: Code for Alice

```
1 repeat forever
2 |   foreach u ∈ N(Alice) do
3 |   |   P_{Alice,u} := ⟨HIu, sign_{Alice}(HIu)⟩
4 |   end
5 |   d_{Alice} := 0
```

3.4.2 Part of algorithm for honest nodes

Now we talk about the more interesting part of the algorithm. We present the part of the algorithm for the honest nodes which is almost strict-stabilizing. That means that they eventually reach a legitimate state.

This is not too bad for us because in our definition of the legitimate state in section 3.1.4 we want that the distance from an honest node v to Alice is between some bounds and not always the shortest possible distance. We can not guarantee that an honest node always finds the actual shortest path to Alice because it could contain a corrupted node. If this corrupted node in the actual shortest path never does anything the honest nodes will never find this path. So the only thing we can guarantee is that the shortest path the honest node v finds is between the actual shortest path and the shortest honest path of v to Alice, if v has an honest path. We will see that our algorithm only changes to a new legitimate state if some honest node v finds a shorter path to Alice. This means that the new legitimate state is better than the old legitimate state. This is the reason it is all right for us that we can change to new legitimate states and only be almost strict-stabilizing instead of fully strict-stabilizing, which is not possible.

The idea of the algorithm is that the honest node v first checks if there exists a valid path from a neighbor to Alice. If yes then v adds itself to the shortest path found and updates its shared memory with all neighbors. Node v also needs to update his distance to Alice to the shortest distance found. As v executes this algorithm forever it updates the distance to Alice every time it finds a shorter path than the current path it has stored.

To check if there is a valid path from a neighbor to Alice an honest node v can check the shared memory $P_{u,v}$ for each neighbor u. Node v then stores the shortest valid path found in P_N. The last thing the honest node v needs to do is to update its shared-memory if it found a new shortest path from a neighbor to Alice. This is done by updating the variables $P_{v,u}$ for each neighbors u. Node v concatenates Hi u to P_N and signs everything with its secret key. Last v needs to update the distance d_v to Alice to the shortest distance found.

We define the function $IsValid$ to check if a path P is valid as follows:

$IsValid(P = \langle P'||HIv, \sigma_v\rangle) : \sigma_v$ valid AND $IsValid(P')$;

$IsValid(\langle HIv, \sigma_{Alice}\rangle) = true$

Algorithm 2: Code for honest node $v \neq$ Alice

```
1  repeat forever
2      // Find current shortest path from neighbors to Alice
3      P_N := null, d_v := ∞
4      foreach u ∈ N(v) do
5          if IsValid(P_{u,v}) then
6              if (P_N = null) OR (#signatures in P_{u,v} < #signatures in
                P_N) then
7                  P_N := P_{u,v} := ⟨p_1, p_2⟩
8              end
9          end
10     end
11
12     // If there is a path from a neighbor to Alice update P_{v,u} for each
       neighbor u
13     if P_N ≠ null then
14         foreach u ∈ N(v) do
15             P_{v,u} := ⟨P_N||HIu, sign_v(P_N||HIu)⟩
16         end
17
18         // Update new shortest distance to Alice
19         d_v := #signatures in p_2
20     end
```

This concludes our solution algorithm, which is composed of Algorithms 1 and 2, for solving the single-source shortest path problem with the predefined conditions.

The last part in this section is to prove that our solution algorithm is almost strict-stabilizing for honest nodes.

Lemma 3.2. *Eventually every honest node v that has an honest path to Alice has d_v between the distance of the shortest path from v to Alice an the distance of the shortest honest path from v to Alice.*

Proof. If an honest node v has an honest path to Alice the upper bound is satisfied because the shortest honest path will be found by Algorithm 2 and d_v is set to the distance of the shortest honest path. From line 6 of Algorithm 2 it follows that v only updates its path if it finds a shorter path. Therefore d_v cannot be smaller than the distance of the shortest path between v and Alice.

If v does not have an honest path, d_v is still not smaller than the shortest path, follows from the fact before. □

Lemma 3.3. *Every honest neighbor v of Alice has $d_v = 1$.*

Proof. This follows from Lemma 3.2 as the shortest and also shortest honest distance from a neighbor of Alice to Alice is 1. □

Lemma 3.4. *Alice has $d_{Alice} = 0$.*

Proof. This follows from line 5 of Algorithm 1. □

The next result follows immediately from Lemmas 3.2, 3.3 and 3.4.

Lemma 3.5. *Eventually our algorithm reaches a legitimate state for honest nodes.*

Finally using the results from above we show that our algorithm is almost strict-stabilizing.

Theorem 3.6. *Our algorithm achieves almost strict-stabilizing single-source shortest path for honest nodes.*

Proof. In Lemma 3.5 we showed that a legitimate state is eventually reached.

To show that we indeed achieve almost strict-stabilization it remains to prove that once a legitimate state is reached the system reaches another legitimate state if it is changed.

Assume that after reaching a legitimate state an honest node gets activated. We denote the first such node by v. This means that v found a shorter path to Alice and updates its d_v. By Lemma 3.2 it follows that the new d_v must also be in bound and the system reaches another legitimate state. □

Chapter 4

Graph Coloring

Graph coloring is another very interesting graph problem. In this section the problem we tackle is to find a coloring in a given graph G so that no two neighbors have the same color. We want to use at most $\Delta + 1$ colors, where Δ is the max degree in the graph. G consists of two different types of nodes: honest nodes and byzantine nodes. In addition transient faults can occur at any time. The goal is to find an algorithm that finds a proper coloring for G and which satisfies the definition of strict-stabilization.

4.1 Defining the problem

4.1.1 System

The system also consists of a given graph G with two types of nodes:

- Honest nodes

  All honest nodes execute the same algorithm. They follow the steps of the algorithm correctly. The goal is that they find a proper coloring so that no two neighbors have the same color.

- Byzantine nodes

  The byzantine nodes are controlled by the central adversary. They can deviate from the steps of the algorithm arbitrarily and try to mislead the honest nodes.

The following holds for our system:

- Transient faults exist

  Transient faults can again reset the states of the nodes. They can change or erase the variables of the nodes, hence also the colors.

Also for this problem there is a daemon. In the next sections we will present three solutions for two different daemons, one for a fair central daemon and two for a distributed daemon, where one is deterministic and the other is randomized. The respective definitions of the daemons will follow in the corresponding section. We also have synchronous communication in our system.

4.1.2 Notations

We use shared memory to store the colors of the nodes. With this method the nodes can always check the colors of their neighbors.

- C_v

  Represents the color of node v.

- $C_{v,u}$

  Node v stores its color C_v in the variables $C_{v,u}$ for each u in $N(v)$, the neighborhood of v, to represent the shared memory.

- $d(v)$

  Defines the degree, number of neighbors, of node v.

- Δ

  Defines the degree of graph G. This corresponds to the highest $d(v)$, highest number of neighbors, in the graph G.

4.1.3 Conditions

For this problem we also define some conditions under which the algorithm has to perform and the the properties the outputs should satisfy.

- Colors are in the set $\{1, 2, \ldots, \Delta, \Delta + 1\}$

  We represent the colors by numbers from the set above. If there are $\Delta + 1$ colors it is always possible to find a proper coloring. This follows from the fact that every node v has at most Δ neighbors, hence there is always a free color for v.

  A color c is free for a node v if no neighbor of v has color c.

4.1.4 Legitimate state

For this problem we have a special case for the legitimate state. The corrupted nodes can always choose the same color as their neighbors and force them to

change their color in order to have a proper coloring. This contradicts the standard definition of strict-stabilization. The standard definition requires that the honest nodes do not change their state, which is composed by the color variables. If the corrupted nodes can always trigger color changes for their neighbors we can never guarantee that honest nodes with byzantine neighbors reach a legitimate state and remain there.

Therefore we define the legitimate state only for honest nodes with distance at least two to byzantine nodes. We define the distance between two nodes as the length of the shortest path connecting those two nodes.

Let V_d denote the set of honest nodes with distance at least d to a byzantine node.

The legitimate state for this problem is:

- $\forall v \in V_2 : \forall u \in N(v) : C_v \neq C_u$, meaning that for all honest nodes in V_2 there is no neighbor with the same color. Nodes in V_2 do not have conflicts.
- $\forall v \in V_1 : C_v \in \{1, 2, \ldots, \Delta, \Delta + 1\}$

4.2 Difficulties encountered

While working on this problem we encountered a few issues which we needed avoid or solve to be able to find a solution algorithm.

Regardless of the coloring algorithm, there are many scenarios in which adjacent nodes can become conflicting, meaning that they choose the same color. For example a transient fault could cause that. We found that fixing such conflicts is not trivial. We encountered issues caused by symmetry and byzantine nodes.

A solution used in well known algorithms is using an ordering based on ids, however corrupted parties can cause honest nodes to wait indefinitely.

4.3 Fair central daemon

We are now going to present a solution for a central and fair daemon. A fair central daemon picks which enabled node should make the next move. We assume that the daemon is controlled by the adversary. However, the daemon must be fair: meaning that no enabled node can wait indefinitely until it is activated.

The fact that the fair central daemon chooses exactly one node to make the next move helps us solving conflicts between neighbors. As mentioned previously choosing which node should change its color first in a conflict represents an issue. In the case with a fair central daemon this is already solved because the daemon chooses which node should make a move.

In our algorithm, each node picks a color only if it is necessary. More specifically, node v is enabled if one of the following conditions hold:

- Its current color is not valid, meaning not in $\{1, 2, \ldots, d(v), d(v)+1\}$: either because of a transient fault or of the initial state.
- It has a conflict.

In either case, when node v is also activated, it picks the lowest free color available.

Note that in the case of a conflict between two adjacent nodes in V_2, only one of the two conflicting neighbors is activated by the fair central daemon, and hence the conflict is solved. This guarantees also that no new conflicts are crated between the nodes in V_2 because they only choose free colors.

Algorithm 3: Code for honest node v

```
1 repeat forever
2     if C_v ∉ {1, 2, ..., d(v), d(v) + 1} OR v has a conflict then
3         C_v := smallest free color
4         foreach u ∈ N(v) do
5             C_{v,u} := C_v
6         end
7     end
```

The last part in this section is to prove that Algorithm 3 is strict-stabilizing for honest nodes in V_2.

Lemma 4.1. *If an honest node v is activated then $C_v \in \{1, 2, \ldots, d(v), d(v)+1\}$.*

Proof. This follows from lines 2 and 3 from Algorithm 3. If an honest node v is activated it sets C_v to the smallest free color and as v has $d(v)$ neighbors there is always a free color for v in $\{1, 2, \ldots, d(v), d(v) + 1\}$. □

Lemma 4.2. *Eventually the nodes in V_2 do not have any conflicts.*

Proof. Assume v from V_2 and u, its neighbor from V_1, have the same color. Given that in such a scenario v and u are honest, they both get enabled, because of line 2 from Algorithm 3. As the daemon is central and fair at least one of the nodes v and u is activated eventually and in line 3 from Algorithm 3 picks a non conflicting color. Hence eventually C_v is different from C_u. □

The next result follows immediately from Lemmas 4.1 and 4.2.

Lemma 4.3. *Eventually Algorithm 3 reaches a legitimate state for nodes in V_2.*

Finally using the results from above we show that Algorithm 3 is strict-stabilizing for nodes in V_2.

Theorem 4.4. *Algorithm 3 achieves strict-stabilizing graph coloring for nodes in V_2.*

Proof. In Lemma 4.3 we showed that a legitimate state is eventually reached.

To show that we indeed achieve strict-stabilization it remains to prove that once a legitimate state is reached the state of the nodes in V_2 remains the same unless a transient fault occurs.

Once a legitimate state is reached, honest nodes, which are not in V_2, won't pick the same color as their neighbors in V_2 by line 3 of Algorithm 3.

Assume that after reaching a legitimate state a node in V_2 gets activated. We denote the first such node by v. By Lemma 4.1 we obtain that v has a conflict and this contradicts that we are in a legitimate state. □

4.4 Distributed daemon

We now change the daemon to a distributed daemon where everything happens at the same time. We consider a synchronous model, meaning that all nodes can make a step at the same time. Hence two neighbor nodes of V_2 can change their color at the same time and potentially create new conflicts if they choose the same color. Another issue with that is if there is a conflict between node v and node u it could happen that both nodes update their color at the same time and the conflict does not get solved.

4.4.1 Deterministic algorithm

In this section we present a deterministic algorithm. To fix a conflict it is necessary that the nodes have a setup to break the symmetry. Therefore in this section we assume that each node v has a unique ID_v. The rule is then that the node with the smaller ID changes first. This is good because then only one conflicting node changes its color and the conflict is solved. But with this method another issue arises. If an honest node v has a conflict with a corrupted neighbor u and $ID_u < ID_v$ then u can force v to wait infinitely long if u does not change its color. Our solution for this issue is that each conflicting node only waits a finite amount of time and if the conflict is still not solved it changes its color anyway. How long should an honest node wait before changing its color? Using an a priori defined amount of time could cause honest nodes to change their color at the same time, and create new conflicts indefinitely. We solve this issue by assigning each node v an unique amount of time: ID_v rounds. IDs are positive natural numbers.

An honest node v will also change its color if the color is not in $\{1, 2, \ldots, d(v), d(v)+1\}$. Node v will always choose the smallest free color available.

Algorithm 4: Deterministic code for honest node v

```
1  counter := 0
2  repeat forever
3  |  if C_v ∉ {1, 2, ..., d(v), d(v) + 1} then
4  |  |  C_v := smallest free color
5  |  end
6  |
7  |  if v has a conflict then
8  |  |  if ID_v is smallest id among conflicting neighbors then
9  |  |  |  C_v := smallest free color
10 |  |  |  foreach u ∈ N(v) do
11 |  |  |  |  C_{v,u} := C_v
12 |  |  |  end
13 |  |  |  counter := 0
14 |  |  end
15 |  |  else
16 |  |  |  counter++
17 |  |  |  if ID_v = counter then
18 |  |  |  |  C_v := smallest free color
19 |  |  |  |  foreach u ∈ N(v) do
20 |  |  |  |  |  C_{v,u} := C_v
21 |  |  |  |  end
22 |  |  |  |  counter := 0
23 |  |  |  end
24 |  |  end
25 |  end
```

Now we prove that Algorithm 4 is strict-stabilizing.

Lemma 4.5. *If an honest node v is activated then $C_v \in \{1, 2, \ldots, d(v), d(v)+1\}$*

Proof. If honest node v is activated it sets C_v to the smallest free color and as v has $d(v)$ neighbors there is always a free color for v in $\{1, 2, \ldots, d(v), d(v)+1\}$. □

Lemma 4.6. *Eventually the nodes in V_2 do not have any conflicts.*

Proof. We consider two cases:

Case 1:

Assume v from V_2 and u, its neighbor from V_1, have the same color. Given that in such a scenario v and u are honest, both v and u get enabled, because of line 7 from Algorithm 4. As ID_v and ID_u are different it follows from line 8 from

Algorithm 4 that the node with the smaller ID will eventually change to a non conflicting color. If v and u have to wait for other nodes with smaller IDs it follows from line 17 from Algorithm 4 that eventually v waits for ID_v rounds or u waits for ID_u rounds and the node with the smaller ID changes to a non conflicting color. If it happens that both nodes v and u change their colors at the same time, because their counters say they should change, it might happen that they create a new conflict. In this case they will start their counters at the same time and the node with the smaller ID will then eventually change to a conflicting color if it waited ID rounds. Hence eventually C_v is different from C_u.

Case 2:

As the counters are not necessarily synchronized because of transient faults it might happen that two neighbors change their color at the same time to the same color and create a new conflict. This cannot happen more than once for a pair of neighbors from V_1. Let $k \cdot ID_v + t_v$ and $k \cdot ID_u + t_u$ denote the rounds when nodes v and u from V_1 change their colors, for some variables k, t_v and t_u, where t_v and t_u represent the starting rounds of the counters. From the fact that $k \cdot ID_v + t_v = k \cdot ID_u + t_u$ happens at most once, because these are linear functions, the scenario above cannot happen more than once.

As in case 1 the conflict eventually gets solved and case 2 can not happen infinitely it follows that the nodes in V_2 eventually do not have any conflicts.

□

The next result follows immediately from Lemmas 4.5 and 4.6.

Lemma 4.7. *Eventually Algorithm 4 reaches a legitimate state for nodes in V_2.*

Finally using the results from above we show that Algorithm 4 is strict-stabilizing for nodes in V_2.

Theorem 4.8. *Algorithm 4 achieves strict-stabilizing graph coloring for nodes in V_2.*

Proof. In Lemma 4.7 we showed that a legitimate state is eventually reached.

To show that we indeed achieve strict-stabilization it remains to prove that once a legitimate state is reached the state of the nodes in V_2 remains the same unless a transient fault occurs.

Once a legitimate state is reached, honest nodes, which are not in V_2, won't pick the same color as their neighbors in V_2 by lines 4, 9 and 18 of Algorithm 4.

Assume that after reaching a legitimate state a node in V_2 gets activated. We denote the first such node by v. By Lemma 4.5 we obtain that v has a conflict and this contradicts that we are in a legitimate state. □

A negative point for this algorithm is that we cannot say when it will reach a legitimate state. The only thing we can say is that it will reach a legitimate state in a finite amount of time. The time however depends on the size of the IDs and therefore how long the nodes are going to wait before changing their color anyway. The longer the IDs the longer can the Algorithm 4 need to reach a legitimate state.

4.4.2 Randomized algorithm

We aim to obtain a better efficiency by using randomization to break the symmetry instead of IDs. For this we adapt Algorithm 3 so that it works under a distributed daemon. There are two main parts which we need to adapt. First if there is a conflict which node has to change, as we do not have the central daemon anymore which solves this for us. The second part is how to choose the new color so that we do not create new conflicts or there is a low probability of creating new conflicts.

To solve this issues we used randomization. The idea is to have the same rules as in Algorithm 3 but, instead of choosing the smallest free color, an honest node v is going to choose a color c uniformly at random from the set $\{1, 2, \ldots, d(v), d(v) + 1\}$. If c is free then v changes its color to c and if c is taken by a neighbor then v does noting and waits for the next round.

Algorithm 5: Code for honest node v

```
1  repeat forever
2      if C_v ∉ {1, 2, ..., d(v), d(v)+1} OR v has a conflict then
3          c := random color from {1, 2, ..., d(v), d(v)+1}
4          if c is free then
5              C_v := c
6              foreach u ∈ N(v) do
7                  C_{v,u} := C_v
8              end
9          end
10     end
```

Randomization helps us with solving the conflicts. As both nodes could change the color at the same time with randomization they will hopefully choose different free colors or only one node will change its color. What we want now is to find an upper bound for the expected number of rounds until all honest nodes in V_2 find a proper coloring. For this we need to find a lower bound for the success probability for an honest node v from V_2 to be safe in the next round. Safe means that v has no conflicts in the next round and it will stay without conflicts because its neighbor nodes will never choose the color of v because it is taken.

We first present a few observations. If color k is the highest free color for v then

it is guaranteed that all colors from $k+1$ to $d(v)+1$ are taken. This means that for each color c from $k+1$ to $d(v)+1$ there is at least one neighbor node of v which has color c. There are $d(v)+1-k$ such nodes. The probability that none of these $d(v)+1-k$ nodes pick color k if v picks k can be lower bounded as follows:

$$\begin{aligned}\prod_{i=k}^{d(v)}\left(1-\frac{1}{i+1}\right) &= \prod_{i=k}^{d(v)}\frac{i}{i+1}\\ &= \frac{k}{k+1}\cdot\frac{k+1}{k+2}\cdot\ldots\cdot\frac{d(v)-1}{d(v)}\cdot\frac{d(v)}{d(v)+1}\\ &= \frac{k}{d(v)+1}\end{aligned}$$

In the first formula we use $i+1$, where i goes from k to $d(v)$, because we know that for k to $d(v)$ there is at least one neighbor that has this degree. This follows from the facts that colors $k+1$ to $d(v)+1$ are taken and a node u has always $d(u)+1$ colors from which it can choose.

The other $k-1$ neighbor nodes of v can have smaller or higher colors than k. If they have a smaller color than k we can not guarantee that they can not choose color k because they could still have a higher degree than k. If they have a higher color than k we know that they can choose color k but we we do not know which degree they exactly have. Therefore we have to assume that all $k-1$ neighbors could potentially choose color k. For them to choose color k they need at least degree $k-1$. Therefore the probability that none of these $k-1$ nodes pick color k if v picks k can be lower bounded as follows:

$$\begin{aligned}\prod_{i=1}^{k-1}\left(1-\frac{1}{k}\right) &= \prod_{i=1}^{k-1}\frac{k-1}{k}\\ &= \left(\frac{k-1}{k}\right)^{k-1}\\ &> \frac{1}{e}\end{aligned}$$

We use $\frac{1}{k}$ in the first formula because the $k-1$ neighbors have k colors from which they can choose but color k is already taken by v. The last step follows from the fact that $\lim_{k\to\infty}\left(\frac{k-1}{k}\right)^{k-1} = \frac{1}{e}$ and it is a decreasing function for $k>1$.

The last observation is that the probability of v to choose a free color is at least the probability to choose the highest free color k. The probability of v to choose k is

$$\frac{1}{d(v)+1}$$

Using these observations, we can now compute a lower bound for an honest node v from V_2 to be safe in the next round. The probability to be safe in the next round is at least the probability to pick a free color times the probabilities that no neighbor chooses the same color. The probabilities of the nodes in V_2 of picking a color are independent, because if node v chooses color c it does not effect the probability of the neighbors of v to choose color c as well. With this fact it follows

$$\begin{aligned} P\left[v \text{ is safe in next round}\right] &\geq \frac{1}{d(v)+1} \cdot \frac{k}{d(v)+1} \cdot \frac{1}{e} \\ &\geq \frac{1}{\left(d(v)+1\right)^2 \cdot e} \\ &\geq \frac{1}{\left(\Delta+1\right)^2 \cdot e} \end{aligned}$$

The first step follows from the fact that probability of v to be safe is at least the probability of v being safe if it chooses the highest free color k. Therefore the probability of v to be safe is at least $\frac{1}{(\Delta+1)^2 \cdot e}$.

Now we can calculate the number of expected rounds until honest node v of V_2 is safe. If $p > \frac{1}{(\Delta+1)^2 \cdot e}$ is the probability that v is safe in the next round then the expected number of rounds until v is safe is $\frac{1}{p}$. Therefore we have

$$\begin{aligned} \mathbb{E}\left[\#\text{ rounds until } v \text{ is safe}\right] &= \frac{1}{p} \\ &< \frac{1}{\frac{1}{(\Delta+1)^2 \cdot e}} \\ &= \left(\Delta+1\right)^2 \cdot e \end{aligned}$$

In conclusion $\left(\Delta+1\right)^2 \cdot e$ is an upper bound for the expected number of rounds until v is safe.

However what we want is an upper bound for the expected number of rounds until all nodes in V_2 are safe and we have a proper coloring. Let say that n is the number of honest nodes in V_2. Then an upper bound can be composed as follows. We could say that one node after the other gets safe. For example first node v gets safe then node u gets safe and so on. From this observation it follows that

$$\begin{aligned} \mathbb{E}\left[\#\text{ rounds until all nodes in } V_2 \text{ are safe}\right] &\leq \mathbb{E}\left[\#\text{ rounds until } v \text{ is safe}\right] \cdot n \\ &< \left(\Delta+1\right)^2 \cdot e \cdot n \end{aligned}$$

We can conclude that in expected $\left(\Delta+1\right)^2 \cdot e \cdot n$ rounds all nodes in V_2 are safe and we have a proper coloring for V_2.

The last part in this section is to prove that Algorithm 5 is strict-stabilizing for honest nodes in V_2.

Lemma 4.9. *If an honest node v is activated it sets C_v to a color from the set $\{1, 2, \ldots, d(v), d(v)+1\}$ in expected $(\Delta+1)^2 \cdot e$ rounds.*

Lemma 4.10. *In expectation the nodes in V_2 do not have any conflicts within $(\Delta+1)^2 \cdot e \cdot n$ rounds, where n is the number of nodes in V_2.*

The next result follows immediately from Lemmas 4.9 and 4.10.

Lemma 4.11. *In expected $(\Delta+1)^2 \cdot e \cdot n$ rounds, n denotes the number of nodes in V_2, Algorithm 5 reaches a legitimate state for nodes in V_2.*

Finally using the results from above we show that Algorithm 5 is strict-stabilizing.

Theorem 4.12. *Algorithm 5 achieves strict-stabilizing graph coloring for nodes in V_2.*

Proof. In Lemma 4.11 we showed that a legitimate state is reached.

To show that we indeed achieve strict-stabilization it remains to prove that once a legitimate state is reached the state of the nodes in V_2 remains the same unless a transient fault occurs.

Once a legitimate state is reached, honest nodes, which are not in V_2, won't pick the same color as their neighbors in V_2 by line 4 of Algorithm 5.

Assume that after reaching a legitimate state a node in V_2 gets activated. We denote the first such node by v. By Lemma 4.9 we obtain that v has a conflict and this contradicts that we are in a legitimate state. □

Bibliography

[1] E. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, pp. 643–644, 1974.

[2] M. Nesterenko and A. Arora, "Tolerance to unbounded byzantine faults," *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pp. 22–29, 2002.

[3] S. Dubois, S. Tixeuil, and N. Zhu, "The byzantine brides problem," in *Fun with Algorithms*, E. Kranakis, D. Krizanc, and F. Luccio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 107–118.

[4] T. Masuzawa and S. Tixeuil, "Stabilizing Link-Coloration of Arbitrary Networks with Unbounded Byzantine Faults," *International Journal of Principles and Applications of Information Science and Technology*, vol. 1, no. 1, pp. 1–13, Dec. 2007. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01152556

[5] T. C. Huang, "A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity," *Journal of Computer and System Sciences*, vol. 71, no. 1, pp. 70–85, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000005000139

[6] A. Kosowski and L. Kuszner, "Self-stabilizing algorithms for graph coloring with improved performance guarantees," in *ICAISC*, 2006.