Iowa Research Online
The University of Iowa's Institutional Repository

University of Iowa
Iowa Research Online

Theses and Dissertations

Fall 2009

Extensions and refinements of stabilization

Anurag Dasgupta
University of Iowa

Copyright 2009 Anurag Dasgupta

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/351

Recommended Citation
Dasgupta, Anurag. "Extensions and refinements of stabilization." PhD (Doctor of Philosophy) thesis, University of Iowa, 2009.
https://ir.uiowa.edu/etd/351.

Follow this and additional works at: https://ir.uiowa.edu/etd


Part of the Computer Sciences Commons

EXTENSIONS AND REFINEMENTS OF STABILIZATION

by

Anurag Dasgupta

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Professor Sukumar Ghosh

ABSTRACT

Self-stabilizing system is a concept of *fault-tolerance* in distributed computing. A distributed algorithm is self-stabilizing if, starting from an arbitrary state, it is guaranteed to converge to a *legal state* in a finite number of states and remains in a legal set of states thereafter. The property of self-stabilization enables a distributed algorithm to recover from a transient fault regardless of its objective. Moreover, a self-stabilizing algorithm does not have to be initialized as it eventually starts to behave correctly.

In this thesis, we focus on extensions and refinements of self-stabilization by studying two non-traditional aspects of self-stabilization.

In traditional self-stabilizing distributed systems [15], the inherent assumption is that all processes run predefined programs mandated by an external agency which is the owner or the administrator of the entire system. The model works fine for solving problems when processes cooperate with one another, with a global goal. In modern times it is quite common to have a distributed system spanning over multiple administrative domains, and processes have selfish motives to optimize their own *payoff*. Maximizing individual payoffs under the umbrella of stabilization characterizes the notion of *selfish stabilization*.

We investigate the impact of selfishness on the existence of stabilizing solutions to specific problems in this thesis. Our model of selfishness centers on a graph where the set of nodes is divided into subsets of distinct colors, each having their own

unique perception of the edge costs. We study the problems of constructing a rooted *shortest path tree* and a *maximum flow tree* on this model, and demonstrate that when processes are selfish, there is no guarantee that a solution will exist. We prove that the complexity of determining the existence of a stabilizing solution is NP-complete, carefully characterize a fraction of such cases, and propose the construction of stabilizing solutions wherever such solutions are feasible.

Fault containment and system availability are important issues in today's distributed systems. In this thesis, we show how fault-containment can be added to *weakly stabilizing* distributed systems. We present solutions using a randomized scheduler, and illustrate techniques to bias the random schedules so that the system recovers from all single faults in a time independent of the size of the system, and the effect of the failure is contained within constant distance from the faulty node with high probability (this probability can be controlled by a user defined tuning parameter). Using this technique, we solve two problems: one is the *persistent-bit problem*, and the other is the *leader election problem*.

Abstract Approved: _____

Thesis Supervisor

_____

Title and Department

_____

Date

EXTENSIONS AND REFINEMENTS OF STABILIZATION

by

Anurag Dasgupta

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Professor Sukumar Ghosh

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Anurag Dasgupta

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2009 graduation.

Thesis Committee: _____
Sukumar Ghosh, Thesis Supervisor

_____
Ted Herman

_____
Alberto Segre

_____
Kasturi Varadarajan

_____
Jon Kuhl

To my parents

ACKNOWLEDGEMENTS

First of all, I am grateful to my advisor, Dr. Sukumar Ghosh for his superior guidance and support throughout my Ph.D. years. I would also like to thank my committee members, Dr. Ted Herman, Dr. Alberto Segre, Dr. Kasturi Varadarajan, and Dr. Jon Kuhl, for their time and support in spite of their busy schedules. I am grateful to Dr. Sébastien Tixeuil and Dr. Johanne Cohen, their many insightful ideas and suggestions have tremendously helped improve the results and they have actively collaborated with us in some of the results presented in this thesis. It has been a real pleasure working with them. I would also like to thank my friend Xin Xiao, a Ph.D. student in the Department of Computer Science at the University of Iowa, for helping me on numerous occasions with the analysis of the results. Her analytical and questioning skills have made the contents of this thesis stronger and clearer. I am also grateful to my family and a few dear friends for their unwavering support over the past years. This thesis may not have been completed without the support of the people mentioned above.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Preamble

A *distributed system* consists of a collection of autonomous computers, connected through a network and distributed system software, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility. This is in contrast to a network, where the user is aware that there are several machines whose locations, storage replications, load balancing, and functionalities are not transparent. The software enables computers to coordinate their activities and to share the resources of the system hardware, software, and data. Airline reservation systems, banking networks, peer-to-peer networks etc. are some typical examples of distributed systems. There are lots of advantages of connecting remote users with remote resources in an open and scalable way. Benefits of distributed systems include bridging geographic distances, tolerating failures and perturbations, improving performance and availability, maintaining autonomy, reducing cost, and allowing for interaction.

To be truly reliable, a distributed system must have the following characteristics:

1. **Fault-Tolerant:** The ability of a system or component to continue normal operation despite the presence of hardware or software faults is termed as fault-tolerance. Fault-tolerance is particularly sought-after in high-availability or

life-critical systems. Failed components should be able to rejoin the system, after the cause of failure has been repaired.

2. **Highly Available:** High availability is a system design protocol and its implementation ensures a certain degree of operational continuity during a given period. Users want their systems to be ready for providing service at all times. Availability refers to the ability of the user community to access the system.

3. **Scalable:** Scalability is generally difficult to define and in any particular case, it is necessary to define the specific requirements for scalability on those dimensions which are deemed important. Scalability can be measured in various dimensions, such as load scalability (the ability for a distributed system to easily expand and contract its resource pool to accommodate heavier or lighter loads. Alternatively, the ease with which a system or component can be modified, added, or removed, to accommodate changing load), geographic scalability (the ability to maintain performance, usefulness, or usability regardless of expansion from concentration in a local area to a more distributed geographic pattern.), administrative scalability (the ability for an increasing number of organizations to easily share a single distributed system.), etc.

4. **Predictable Performance:** A system's predictability is defined as the ability to provide desired responsiveness in a timely manner.

5. **Secure:** A secure system must have control of access to its resources and services, especially of its data and operating system files.

Achieving all of the above goals/characteristics is quite challenging. Probably one of the most difficult challenges in a distributed system is the requirement of it to be able to continue operating correctly even when failures occur in the system. Failures or faults can be broadly divided into two categories: hardware and software.

Hardware faults were a dominant concern in system reliability until the late 80's. Hardware reliability has improved enormously during the last two decades. Decreased heat production and power consumption of smaller circuits, reduction of off-chip connections and wiring, and high-quality manufacturing techniques have all played a positive role in improving hardware reliability. Hardware faults are still a major issue in the domain of VLSI fabrication.

Software faults are primarily of two types: a) Permanent design faults which are almost deterministic in nature. Most of these can be identified and weeded out during the testing and debugging phase (or early deployment phase) of the software life cycle. b) Some software failures are temporary internal faults and are intermittent. These are known as *Heisenbugs*. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible. Hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted. Some typical situations in which these types of faults might surface are boundaries between various software components, improper or insufficient exception handling, and interdependent timing of various events. Most recent studies on failure data have reported that a large proportion of software failures are transient in nature, caused by phenomena such as overloads or timing and exception errors.

The distributed nature of the system can make it harder to diagnose and rectify faults. It is therefore desirable that some means of fault tolerance be incorporated in the system so that the system will behave in a desired manner even in the presence of certain types of faults. One part of this thesis investigates the design of a class of fault-tolerant distributed systems.

Informally, a fault-tolerant system specifies two things (i) a fault model to describe the different types of faults and (ii) the type of tolerance. Faults can be classified into various types by considering the behavior or the impact of the faulty component. Some common types of faults in such a classification are:

1. **Halting faults:** A component simply stops and the change is irreversible. There may be various reasons why such a failure occurs. In synchronous systems, halting failure can be detected using timeout within a bounded time, since the faulty component either stops sending heartbeat messages or fails to respond to requests within a specific time period. Computer freezing is a halting failure.

2. **Omission faults:** Failure to send a scheduled message or receive messages that have been sent due to transmission media problems, internal hardware or software failures, or shortage of buffer space, results in a message being discarded with no notification to either the sender or the receiver. For example, this can happen when routers become overloaded.

3. **Network faults:** The functionality of a network can be impaired because

of component malfunction or natural or human-caused disasters (like security attacks), or by component mobility. Partial failures include degradation (and graceful degradation). Due to loss of power or link, a network may also partition into two or more disjoint sub-networks within which messages can be sent, but between which messages cannot be exchanged.

4. **Timing faults:** Timing failure is a failure of a process, or a subsystem of a synchronous real-time system to meet deadline limits set on execution time, message delivery, or clock skew. Asynchronous distributed systems cannot be said to have timing failures as guarantees are not provided for response times. For example, lack of synchronization among clocks on different computers can lead to a timing fault.

5. **Transient Fault:** Faults that are triggered by events of a transient nature are termed as transient faults. For example, electromagnetic radiation, cosmic rays, or other forms of electrical noise can alter the status of one or more processes in processor chips, and can trigger transient faults.

6. **Software Fault:** The inability of a program to continue processing or producing correct outputs due to erroneous logic is called software failure. According to Brocklehurst et al. [5], "a software failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service".

7. **Byzantine faults:** It is an arbitrary fault that can occur during the execu-

tion of a distributed systems algorithm.This captures every conceivable type of faulty behaviors including data corruption or loss, failures caused by malicious programs, etc.

With respect to a given set of faults, the type of tolerance exhibited by a fault-tolerant system can be classified as *masking* or *non-masking*. *Masking* fault-tolerance guarantees that programs continually satisfy their specification in the presence of faults. By way of contrast, *non-masking* fault-tolerance allows the specifications to be temporarily violated, but guarantees that after faults stop occurring, eventually the program executions help the system converge to configurations from where the system continually (re)satisfies their specification.

One approach to incorporate fault-tolerance in distributed systems is to design distributed protocols that guarantee a pre-specified behavior in the presence of certain types of faults. One part of this thesis focuses on a specific class of non-masking fault-tolerance called *self-stabilization*.

1.2 Self-Stabilization

Self-stabilization is a particularly robust and appealing model of fault-tolerance for distributed computation. The subset of desirable states to which the system converges to is called the set of *legal states*. A state not belonging to the set of legal states is called an *illegal state*. A system is self-stabilizing if and only if it satisfies two properties: a) starting from any state, it is guaranteed that the system will eventually reach a legal state (*convergence*), and b) given that the system is in a legal state, it is

guaranteed to stay in a legal state, provided that no fault happens (*closure*) [1]. Thus, a self-stabilizing system will spontaneously recover from one or more transient faults (that take the system to some arbitrary configuration) without the need for manual intervention, as long as no further faults occur. This also implies that self-stabilizing systems do not need initialization, and can be spontaneously deployed.

E.W. Dijkstra in 1974 [15] presented the first self-stabilizing algorithm, in which he coined the phrase and showed the feasibility of designing such protocols in spite of distributed control. This brief paper went largely unnoticed until Leslie Lamport's invited talk at PODC (Principles of Distributed Computing) 1983 [36], where he said: "I regard this as Dijkstra's most brilliant work — at least, his most brilliant published paper ... a milestone in work on fault-tolerance ... I regard self-stabilization to be a very important concept in fault-tolerance, and to be a very fertile field for research." The ability to recover without external intervention is very desirable in modern computer and telecommunications networks, since it would enable them to eventually repair the effect of faults and return to normal operations on their own. Self-stabilization research so far produced numerous theoretical results some of which can be potentially used in practice. In the areas of sensor and peer-to-peer networks, self-stabilizing protocols have been designed for a variety of problems [43, 7, 31, 42, 4, 46, 29]. Topology update, clock synchronization, multiprocessor synchronization, network flow control, graph algorithms, etc. are some of the areas where the concepts of self-stabilization have been widely used. A survey of some of the basic issues of self-stabilization and self-stabilizing protocols designed for various

problems can be found in [27]. Dijkstra's paper received the 2002 ACM PODC Influential Paper Award, one of the highest achievements recognized by the distributed computing community.

A variant of the classic self-stabilization problem is *weak stabilization*, introduced by Gouda [22]. The solutions to some of the problems that we present in this thesis are weakly stabilizing. Weak stabilization guarantees that starting from an arbitrary configuration, there exists at least one computation that leads the system to a legal configuration.

1.3 Limitations of Self-Stabilization

In spite of many desirable features, traditional self-stabilizing protocols suffer from a few limitations. These limitations motivate the current research. The first limitation is that a self-stabilizing protocol only guarantees eventual convergence to some legal state but may behave arbitrarily during convergence to such a desired state. This may not be acceptable for many applications where some minimum guarantee on system behavior is required at any point of time for reasons of safety or availability of the system. Some researchers have addressed such issues. Dolev and Herman [16] suggested a *superstabilizing* protocol, where a passage predicate is satisfied by the global states of the system during convergence from a topology change, and thus it restricts the arbitrary behavior of the system during convergence. A superstabilizing protocol is a protocol that (i) is self-stabilizing, meaning that it can recover from an arbitrarily severe transient fault; and (ii) can recover from a local transient fault while

satisfying a passage predicate during recovery. Gouda and Schneider [21] present a self-stabilizing protocol for the construction and maintenance of a maximum flow tree of a graph, which ensures that during convergence to a correct maximum flow tree from specific initial states, a tree structure is maintained. However, the tree may not be a maximum flow tree until a legal state is reached. Such examples are abundant in self stabilizing distributed systems. A self-stabilizing protocol does not distinguish between an arbitrary initial state and an initial state that is very close to a legal state but not legal. By "close to a legal state" but not legal, we mean a state from which a legal state can be reached by changing the local states of a small number of processes of the system. Ideally the recovery should take place following a short path in the state space to the nearest legal state. However, most of the current self-stabilizing protocols may not follow the shortest path to recovery. In present day distributed systems, the problem arising out of this limitation is highly important. As a consequence, sometimes the recovery from even a single component failure in the system may require a large amount of time, and can potentially contaminate the entire network. This recovery time may be as large as the time to recover from the failure of an arbitrarily large number of components, even though in a legal state such single faults are clearly more likely than an arbitrary number of faults. The remedy to this problem has been addressed in the research on fault-containment [18, 19, 17, 20].

Another limitation is that, current research on the design of self-stabilizing (*a.k.a.* stabilizing) distributed systems assumes that all processes run predefined programs mandated by an external agency who is the owner or the administrator of

the entire system. The model is acceptable only when processes cooperate with one another, and the goal is purely a global one. The model falls apart when the distributed system spans over multiple administrative domains or processes have private goals too. On Internet-scale distributed systems, each process or each administrative domain may have selfish motives to maximize its own *payoff*. In fact, payoffs or cost functions have been the major driving force behind *game theory*, but individual payoffs never figured into the realm of stabilizing distributed systems. There are many applications where individual payoffs are relevant, but the spirit of competition need not conflict with the general spirit of cooperation that is the driving force behind stabilizing algorithms. To clarify this issue, consider that a system of n processes for which a legal configuration is any element of the set of configurations $\{L_1, \cdots, L_k\}$, but different processes have different preferences about their ideal legal configurations. Attaining the individual goal may be possible via the use of asymmetric cost functions that are statically defined, or by the use of specific strategies that may be adopted at run time. Such strategies refine the basic move. For example, to execute the step of choosing a neighbor, different processes may adopt different strategies for choosing the neighbor. While the choice will impact the payoffs, it will not affect the global goal, or the stabilization mechanism. As an example, consider the stabilizing token circulation protocols [15] that have been widely studied by the stabilization community. If there are several classes of processes with competing interests, then in addition to the common goal of reducing the number of tokens to one, each class may try to retain the token among themselves more often than their competitors.

In this context, we introduce the notion of *selfish stabilization*, and address two specific problems in this domain. Maximizing individual payoffs under the umbrella of stabilization characterizes the notion of *selfish stabilization*.

1.4 Selfish Stabilization

Selfish stabilization blends game theory with self-stabilization. There are some strong similarities between the two paradigms, but there are significant differences too. Considering the *players* in games to be equivalent to *processes* in a stabilizing system, the equilibrium in games is comparable to the legal configuration of stabilizing systems, in as much as both satisfy the condition of convergence and closure. However, unlike stabilizing systems, games start from predefined initial configurations, and largely ignore faulty moves or transient state corruptions. An exception is the notion of *bounded rationality* [44] that suggests that economic agents sometimes use heuristics to make decisions rather than a strict rigid rule of optimization in light of the complexity of the situation. In the context of distributed systems, the anarchic behavior of processes for meeting selfish goals can be viewed as a weaker version of byzantine failure.

Stabilizing distributed systems expect all the component processes to run predefined programs that are externally mandated. In Internet scale systems, this is unrealistic, since each process may have selfish interests and motives related to maximizing its own payoff. In this thesis, we formulate the problem of selfish stabilization that shows how competition blends with cooperation in a stabilizing environment.

It is a well known principle of economics that when members of society do not coordinate and instead act independently in a self-interested fashion, the resulting outcome typically does not achieve the maximum possible total social welfare. In our setting, self-interested nodes or simply called, selfish nodes, seek to maximize their own private welfare by making choices available from their immediate neighborhood. In a sense, these choices are greedy as we assume that a selfish node cannot predict or foresee any other move that may eventually lead to a better payoff for it. It is possible that minor restrictions or adjustments of a class of nodes' algorithm may lead to dramatically better outcomes, but in this thesis, we stick to the notion of our above mentioned 'selfish' property.

1.5 Fault Containment

Improved reliability of system components implies that in a well-designed system, the possibility of a massive failure is nominal, and single transient failures are much more likely to occur. The problem of containing the effect of minor failures is becoming important not only because they are more likely to occur, but also due to the dramatic growth of network sizes. Such limited transient faults are likely to give rise to states that are almost legal, rather than states that are arbitrary. Hence it is desirable that from such limited transient faults, the system should recover quickly. To increase the efficiency of fault-tolerance, it is important to guarantee a much faster recovery from all single failures, while also guaranteeing eventual recovery from more major failures. Moreover, to ensure that non-faulty processes largely remain

unaffected by such minor failures, it is important that only a small part of the network around the faulty components make observable state changes that can affect the operations in the application layer.

The tight containment of the effect of single failures depends on the context: *containment in time* implies that all observable variables of the system recover to their legal values in $O(1)$ time, whereas *containment in space* means that the processes at $O(1)$ distance from the faulty process make observable changes. For optimal performance, both of these properties should hold.

Fault-containment has so far largely been addressed in the context of classical self-stabilizing systems. In this thesis, we study fault-containment in weakly stabilizing systems.

1.6 Related Work

Selfish stabilization shares many common properties with game theory as competition and cooperation simultaneously exist in both. Game theory so far has been a hotbed of activities in computational economics (like auctions) and algorithm design. It has also received attention in understanding the intricacies of interdomain routing protocols like BGP. The *stable path problem* [24] is an abstraction of the basic functionality of the Internet's BGP routing protocol, and reflects the instabilities observed in BGP. In the stable path problem, each process has to choose the best path according to some local routing policy, and conflicts between local interests can lead to unstable or oscillating behavior. Cobb et al. [9] proposed a stabilizing solution to

the stable path problem. Halpern [26] presented a perspective game theory for distributed systems researchers. He considers issues in distributed computation relevant to game theorists and in particular, focuses on (a) representing knowledge and uncertainty, (b) dealing with failures, and (c) specification of mechanisms. Moscibroda, Schmid, and Wattenhofer [38] studied the formation of the topology of a P2P network by selfish peers, and presented a negative result that the topology construction may not terminate due to selfish behavior, even if churns are absent. They established the complexity of Nash equilibria in the game theoretic model of P2P networks, and also analyzed [39] the impact of allowing some processes to be malicious or byzantine, whereas others are selfish, and computed the price of malice. Keidar et al.'s Equicast protocol [33] for multicast in a peer-to-peer network deals with freeloaders by treating the system as a non-cooperative game, where nodes are selfish but rational. Each user chooses its own strategy regarding its level of cooperation to participate in the multicast so as to minimize its own cost. The goal of each node is to receive all the multicast packets while minimizing its sending rate. Mavronicolas [37] used a game-theoretic view to model security in wireless sensor networks as a game between the attackers and the defenders. Cao et al. [6] presented a variation of the pursuer-evader games for asset protection using a large-scale sensor-actuator network. The last two are only tangentially related to our work, since stabilization is not an issue.

In the area of fault-containment, for specific problems, self-stabilizing protocols have been studied in the past. In [17], Ghosh et al. solved the leader election problem on an oriented ring, and present a simple self-stabilizing leader election protocol that

recovers in O(1) time from a state with a single transient fault on oriented rings. Only the faulty node and its two neighbors change their states during convergence to a stable state. Thus, the effect of a single fault is tightly contained around the fault. In [18], Ghosh et al. presented a fault-containing self-stabilizing protocol for constructing spanning tree in an arbitrary network. In [20], the same authors demonstrated a fault-containing self-stabilizing protocol for BFS tree construction in an arbitrary network. Kutten and Peleg [35] introduced a protocol for fault-mending that corrects a system from minor failures, but provides no guarantee for stabilization. This paper introduced the concept of fault-local algorithms, which are algorithms whose cost depends only on the (unknown) number of faults. Ghosh et al. [18, 19] and Gupta [25] demonstrated how containment can be combined with stabilization, and analyzed the cost of it. Dolev and Herman introduced superstabilizing protocols [16], that, in addition to being stabilizing, guarantee that during convergence from configurations that arise from legal states by small-scale topology changes, certain passage predicates are satisfied. Herman's self-stabilizing protocol [28] for mutual exclusion investigates the possibility of superstabilizing protocols for mutual exclusion in a ring of processors, where a local fault consists of any transient fault at a single processor, and the passage predicate for mutual exclusion specifies that there be at most one token in the ring, with the single exception of a spurious token co-located with the transient fault. Kutten and Patt-Shamir [34] proposed an asynchronous stabilizing algorithm for the persistent-bit problem. Their solution leads to recovery in O(k) time from any k-faulty configuration. A similar protocol for the problem

of token passing can be found in [3], where Beauquier et al. studied the scenario where up to k faults hit nodes in a reactive asynchronous distributed system by corrupting their state undetectably. In [2] Beauquier et al. investigated k-strong self-stabilizing systems, which satisfy the properties of strong confinement and of k-linear time adaptivity. Strong confinement means that a non faulty processor has the same behavior with or without the presence of faults elsewhere in the system. k-linear time adaptivity means that after k or less faults hitting the system in a correct state, the recovery takes a number of rounds linear in k.

In [22], Gouda introduced weak stabilization. Although this property is strictly weaker than the well-known property of stabilization, weak stabilization is superior to stabilization in several respects and plays an important role in the solutions we provide in this thesis. In [14], Devismes et al. investigated the relative power of weak, self, and probabilistic stabilization, and provided a deterministic weak-stabilizing leader election algorithm on a tree network using a strongly fair scheduler.

1.7 Contributions

This thesis addresses two different problems in the domain of self-stabilization.

In the first, we study the problem of selfish stabilization, and investigate the impact of selfishness on the existence of stabilizing solutions to specific problems. Our model of selfishness centers on a graph $G = (V, E)$ where the set of nodes V is divided into subsets of distinct colors, each having their own unique perception of the edge costs. One of these nodes is the root node. We study the problems of

constructing a rooted *shortest path tree* and a *maximum flow tree* on this model, and demonstrate that when processes are selfish, there is no guarantee that a stabilizing solution will exist. We demonstrate that the complexity of determining the existence of a stabilizing solution is NP-complete. We carefully characterize a fraction of such cases, and propose the construction stabilizing solutions wherever such solutions are feasible [10, 11].

In the second problem, we show how fault-containment can be added to *weakly stabilizing* distributed systems. We present solutions using a randomized scheduler, and illustrate techniques to bias the random schedules so that the system recovers from all single faults in a time independent of the size $n = |V|$ of the system, and the effect of the failure is contained within $O(1)$ distance from the faulty node with high probability (this probability can be controlled by a user defined tuning parameter). Using this technique, we solve two problems: one is the *persistent-bit problem*, and the other is a *leader election problem* on a line topology. Our solutions exhibit a low *fault-gap* and guarantee *high system availability*, where once the output variables of the system recover to a legal configuration, it is immediately ready to handle the next single fault with the same degree of efficiency [12, 13].

1.8 Organization of the thesis

The rest of this thesis is organized as follows. Chapter 2 describes the models and notations of selfish stabilization and fault-containment. It also contains some important definitions. In Chapter 3 and Chapter 4, we study the problem of selfish

stabilization. In Chapter 3, we study the problem of constructing a rooted shortest path tree, whereas in Chapter 4, we study the problem of rooted maximum flow tree formation in the context of selfish stabilization. In the subsequent two chapters, we add fault-containment to weakly stabilizing distributed systems. In Chapter 5, we show a fault-containing solution on the persistent-bit problem, whereas in Chapter 6, we demonstrate a solution on the leader election problem. Finally, Chapter 7 summarizes the results of the thesis and discusses the scope for further work in the area.

CHAPTER 2
MODEL AND NOTATIONS

2.1 Introduction

In the previous chapter, we presented an informal description of the selfish stabilization and fault-containment problems. In this chapter, we present a more formal description of the these problems, and introduce the model, notations and definitions. Let $G = (V, E)$ denote the topology of a distributed system, where V represents the set of processes $\{0, 1, \cdots, n-1\}$, and each edge $(i, j) \in E$ represents a bidirectional link between processes i and j. We use the notation $N(i)$ to represent the neighbors of i: thus $(i, j) \in E \Leftrightarrow j \in N(i)$. Processes communicate with their immediate neighbors (also called the distance-1 neighbors) using the *shared memory* model. The execution of the protocols is organized in steps. In each step, a process i executes a program that consists of one or more guarded actions $g \rightarrow A$, where g is a predicate involving the variables of i and $N(i)$, and A is an action that updates one or more variables of i. The global state or configuration of the system is a tuple consisting of the local states of all the processes. Unless otherwise stated, a deterministic central scheduler (also called a *central demon*) serializes all guarded actions. In some cases, we assume a randomized scheduler, where the central demon *randomly chooses* an action with an enabled guard with uniform probability. A *computation* of the system is a finite or infinite sequence of global states that satisfies two properties: (a) if s and s' are two consecutive states in the sequence, then there exists a process i such

that i has an enabled guard in s and execution of the corresponding action results in the state s', and (b) if the sequence is finite, then in the last state of the sequence, no process has an enabled guard.

Definition 1 (Weak Stabilization)**.** Weak stabilization *[22] guarantees that starting from an arbitrary configuration, there exists* at least one computation *that leads the system to a* legal *configuration.*

2.2 Selfish Stabilization

Given a graph $G = (V, E)$, assume that there are p ($p > 1$) different subsets (or classes or colors) of nodes. We divide V into p disjoint subsets of nodes $V_1, V_2, V_3, \ldots, V_p$, such that $V = \cup_{i=1}^{p} V_i$. For each subset or color, there is a separate cost function that maps the set of edges to the set of positive integers. Starting from an arbitrary initial configuration, the different classes of nodes cooperate with one another to form a rooted tree, and at the same time compete against each other to minimize their cost of communication (individual payoff) with the root node. So, the private goal for every node v is to optimize its cost function, i.e., $cost(v)$.

The goal configuration corresponds to an equilibrium configuration such that no process has an eligible action that can unilaterally decrease its own cost. We define the following two terms first to define an *equilibrium* in the system.

Definition 2 (Stable node)**.** *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \rightarrow N^p$. Let T be a tree of G rooted at r. The node $v \in V_i$ is* stable in T for metric w *if and only*

if it satisfies the following condition:

$$\forall z \in N(v), \sum_{e \in T_{v \to r}} w_i(e) \leq w_i((v,z)) + \sum_{e \in T_{z \to r}} w_i(e)$$

In other word,

$$\forall z \in N(v), w_i(T_{v \to r}) \leq w_i((v,z)) + w_i(T_{z \to r})$$

Similarly, a tree T is stable if every node in T is stable.

Definition 3 (Stable tree)**.** *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \to N^p$. Let T be a tree of G rooted at r. The tree T is* stable for metric w *if and only if for every node $v \in V$, v is stable in T for metric w.*

Definition 4 (Equilibrium)**.** *Let $G = (V, E)$ be a graph and r a node of V. If there exists a* stable tree T *in G rooted at r, then we say that the graph G has a* rooted equilibrium *(in general, we call it an equilibrium unless otherwise stated). A* non-rooted equilibrium *consists of a subset of nodes in G and it does not involve r.*

When a rooted tree T exists in G, this tree defines a unique path $T_{v \to u}$ between any two nodes v and u of V. A node v is stable in a tree T if it has no incentive to chose another neighbor $z \in N(v)$, *i.e.* choosing z would not lower its cost.

We allow processes to choose different algorithms, or switch algorithms from a set $\Sigma = \{P_1, P_1, \cdots, P_k\}$ (each P_i reflects a distinct algorithm) in order to meet their private goals. This is unlike traditional stabilization algorithms where all processes execute the same algorithm. Thus, each state transition (S_i, S_{i+1}) is caused by an action of some algorithm $P_j \in \Sigma$. We will call Σ the *strategy space*.

To devise a stabilizing mechanism for the construction of a *stable tree* w.r.t. a particular metric, the computation should lead to a tree configuration so that the conflicting private goals of local cost optimization do not interfere with the common goal of tree formation.

2.3 Fault-Containment

A stabilizing system converges to a legal configuration LC that is traditionally defined in terms of the observable or *primary* variables. However, in most cases, fault-containment requires the use of auxiliary or *secondary variables* too. We define the local state of each process i as an ordered pair $\langle p_i, s_i \rangle$, where p_i denotes the primary variables, and s_i denotes the secondary variables. Correspondingly, we write the global state as an ordered pair $\langle p, s \rangle$, where p is the collection of all primary variables and s is the collection of all secondary variables. For any global state $\langle p, s \rangle$, $\langle p, s \rangle \in LC \Rightarrow p \in LC_p$ and $s \in LC_s$. The following are some important definitions relevant to the fault-containment problem-

Definition 5 (Stabilization Time)**.** *The* stabilization time *is the maximum time needed to establish LC from an arbitrary initial configuration.*

Though $LC \Rightarrow p \in LC_p$ and $s \in LC_s$, in most cases, it is adequate to establish LC_p only, since the application is not affected.

Definition 6 (Fault-Containment in Time)**.** *The* containment time *means that starting from any single-fault configuration, LC_p is restored in $O(1)$.*

This definition may appear strict in some cases. So we define the notion of

weak fault-containment.

Definition 7 (Weak Fault-Containment)**.** *A system is said to be weakly fault containing if the containment time of the system is independent of n, i.e., the size of the network.*

Note that, for weak fault-containment LC_p does not necessarily have to be restored in $O(1)$ time.

Definition 8 (Fault-Containment in Space)**.** *Containment in space means that starting from a single fault configuration, the primary variables of the processes at* $O(1)$ *distance from the faulty process make observable changes.*

Ideally attempt must be made to restrict the effect of the fault within the immediate neighborhood of the original faulty node, which is the notion of tight containment. This way apart from the faulty region, the rest of the system can still perform its normal operation. The fault should not contaminate a large portion of the network.

Definition 9 (Contamination Number)**.** *The* contamination number *is the maximum number of processes that change the primary part of their local states during recovery from any single-fault configuration.*

Ideally the contamination number should be as small as possible.

Definition 10 (fault-gap)**.** *The* fault-gap *is the worst case time, starting from any single-fault state, to reach a state in* LC*.*

fault-gap determines the availability of the fault-free system, and a low fault-gap is desirable as it reflects better availability. Scale-free fault-gap is an important design goal. If the next failure hits the system sooner than the time duration of the fault-gap, then the guarantee of constant time recovery does not hold anymore. In our solutions, after the primary variables stabilize, the application can resume its normal operation (even if the secondary variables have not stabilized yet) and the system is ready to handle the next fault. So, for our proposed solutions, the *fault-gap* is the worst case time, starting from any single-fault state, to reach a state in LC_p.

2.4 Discussions

Selfishness is not the only challenge in today's distributed systems. Often, modern systems have to cope with malicious Byzantine adversaries who seek to degrade the utility of the entire system independently of their own cost. Especially in the area of security and distributed computing, researchers have devised solutions to defend against such possible attacks [39].

One important metric in this context is the "Price of Malice" of selfish systems [41]. The "Price of Malice" is a ratio that expresses how much the presence of malicious players deteriorates the social welfare of a system consisting of selfish players. Formally, the "price of Malice" is the ratio between the social welfare or performance achieved by a selfish system containing a number of Byzantine players, and the social welfare achieved by an entirely selfish system.

CHAPTER 3
SELFISH STABILIZATION IN SHORTEST PATH TREE

3.1 Introduction

In traditional self-stabilizing distributed systems [15], the inherent assumption is that all processes run predefined programs mandated by an external agency which is the owner or the administrator of the entire system. The model is widely acceptable in the stabilization community and works fine for solving problems when processes cooperate with one another, with a goal that is purely a global one. But what if the processes have some private goals besides the common global goal? It is quite realistic and common in modern times to have a distributed system spanning over multiple administrative domains or processes in the system having private goals. On Internet-scale distributed systems, each process or each administrative domain may have selfish motives to optimize its own *payoff*, but this spirit of competition need not conflict with the general spirit of cooperation that is the driving force behind stabilizing algorithms. This chapter formulates the problem of selfish stabilization, studies the impact of selfishness on the stabilization mechanism, and provides examples to explore how competition blends with cooperation in a stabilizing environment.

Given a graph $G = (V, E)$, assume that there are p ($p > 1$) different subsets (or classes or colors) of nodes. For each subset or color, there is a separate cost function that maps the set of edges to the set of positive integers. Starting from an arbitrary initial configuration, the different classes of nodes cooperate with one another to

form a *shortest path tree* rooted at a designated node, and at the same time compete against each other to minimize their cost of communication with the root node. The communication cost may depend on various factors: for example, ownership of the routers may be a factor in determining the cost of routing traffic for any class of nodes. The processes are free to choose a strategy (an algorithm for reaching their goal) from a given set of strategies, and even switch strategy to satisfy their individual needs. We examine strategies under which, starting from an arbitrary initial configuration, these classes of processes can stabilize to an equilibrium configuration after which no process can unilaterally decrease its communication cost to the root.

3.2 Contributions

We show the strong result that even with two different classes of nodes, no equilibrium configuration may exist. We also prove that determining whether a particular setting admits an equilibrium is NP-complete. In case it is known that a solution to the problem exists, we present a (weakly) stabilizing distributed algorithm to construct a rooted shortest path tree. Alternative strategies for participating nodes are also discussed.

3.3 Model and Notation

We follow the general model of selfish stabilization we introduced in Chapter 2. In addition to that, for the *shortest path tree* problem, we consider the following specific notations:

All nodes in V have a common goal: starting from an arbitrary initial con-

figuration, they collaborate with one another to form a shortest path tree with a given node designated as the root. We divide V into p disjoint subsets of nodes $V_1, V_2, \ldots, V_p$, such that $V = \cup_{i=1}^{p} V_i$

For the ease of description, we associate a distinct color with each subset. In addition to the common goal, these subsets have their own agenda: we call them *private* goals. The private goals of the p different subsets may be conflicting - for example, these subsets of nodes may want to split a common resource. To illustrate such private goals, we convert G into a multi-weighted graph by defining a cost function w of $E \rightarrow N^p$, where N is the set of (strictly) positive integers. For every $i \in [1 \ldots p]$, the function $w_i : E \rightarrow N$ denotes the cost of using edge e,

$$\forall e \in E,\ w(e) = (w_1(e), w_2(e), w_3(e), \ldots, w_p(e))$$

When a shortest path tree T exists in G, this tree defines a unique path $T_{v \rightarrow u}$ between any two nodes v and u of V. The cost of this path for a node belonging to a particular subset V_i is

$$w_i(T_{v \rightarrow u}) = \sum_{e \in T_{v \rightarrow u}} w_i(e)$$

In this chapter, we study the cost of reaching the root r of a shortest path tree T from a node v belonging to V_i. This cost is equal to $cost(v) = w_i(T_{v \rightarrow r})$.

The private goal for every node v is to minimize $cost(v)$. A node v is stable in a tree T if it has no incentive to chose another neighbor $z \in N(v)$, *i.e.* choosing z would not lower its cost.

3.4 Non-existence of Equilibrium: An Example

The roadmap. Among the set of all such problems, there are cases where no equilibrium exists, regardless of the strategy chosen by the processes. This is tantamount to presenting instances of metrics for which no stable tree exists. Fig 3.1 shows one such example with a 2-partition of the nodes of a graph of 5 nodes (called graph $\mathcal{Z}$). The set of nodes V is split into V_1 (the white nodes) and V_2 (the black nodes): $V_1 = \{r, a, b\}$, $V_2 = \{x, y\}$. The metric w is described in Fig 3.1. Let us show that there exists no stable tree here: assume there exists a stable tree T. Now, if edge (x, r) is in T, this implies that edge (x, a) is also in T (otherwise T would not be stable because the shortest path between a and r for metric w_1 is (a, x, r)). Edge (a, y) is in T because path (y, a, x, r) is a shortest path between node y and r for metric w_2. Now consider node b. Edges (b, x) and (b, y) cannot be in T. Thus, edge (b, r) must be in T. So this contradicts the stable property of T because at node x, we would have $\sum_{e \in T_{x \to r}} w_2(e) = 3 \geq w_2((x, b)) + \sum_{e \in T_{b \to r}} w_i(e) = 2$. The same argument can be repeated for edges (a, r), (b, r), and (y, r) by assuming one of them is in T. So there exists no stable tree for this instance of w. Of course, this example can be generalized to an arbitrary $p \geq 2$.

Clearly, there is no point in searching for a stabilizing solution, because no such solution can be found. We now demonstrate that deciding whether there exists an equilibrium is an NP-complete problem.

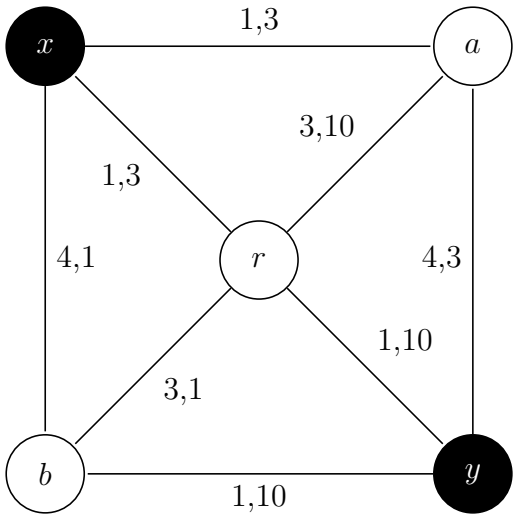

Figure 3.1: The graph $\mathcal{Z}$ with 5 nodes

3.5 The Decision Problem: Does Equilibrium Exist?

In this section, we discuss the existence of an equilibrium (*i.e.* a stable tree T for a particular metric w) in relation to the number of conflicting interests (*i.e.* the number p in the partition of nodes of V into p sets $V_1, V_2, \ldots, V_p$). Recall that the special case of $p = 1$ is the usual problem of the shortest path tree with non zero edge costs, so an equilibrium always exists. We first prove that when $p \geq 2$, determining whether there exists an equilibrium for a particular metric w is an NP-complete problem.

Theorem 1. *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with $p \geq 2$ elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \rightarrow N^p$. The problem of deciding the existence of a stable shortest path tree T for metric w is NP-complete.*[1]

[1]This theorem was proved with help from Johanne Cohen and Sébastien Tixeuil

Proof. It is easy to show that this problem is in NP. Indeed, given a particular tree T, checking whether T is stable for metric w can be done in polynomial time by checking if every node is stable in T for metric w.

Then, we construct a polynomial transformation (almost similar to transformation described in [24]) with the NP-complete problem 3-SAT defined as follows:

Instance: Collection $C = \{c_1, c_2, \ldots c_m\}$ of clauses on a finite set U of variables such that $|c_i| = 3$ for $1 \leq i \leq m$.

Question: Is there a truth assignment for U that satisfies all the clauses in C?

We now present the transformation $\mathcal{R} : (C, U) \rightarrow (G, r, \mathcal{P}, w)$:

1. $V_1 \leftarrow \{r\}$; $V_2 \leftarrow \emptyset$; $E \leftarrow \emptyset$;
2. for each variable $v \in U$ do:
   (a) $V \leftarrow V \cup \{v, \overline{v}\}$;$V_1 \leftarrow V_1 \cup \{v\}$;$V_2 \leftarrow V_2 \cup \{\overline{v}\}$;
   (b) $E \leftarrow E \cup \{(v, r), (\overline{v}, r), (v, \overline{v})\}$;
   (c) $w(v, r) \leftarrow (3, 1)$; $w(\overline{v}, r) \leftarrow (1, 3)$; $w(v, \overline{v}) \leftarrow (1, 1)$;
3. for each element $c \in C$ do:
   (a) $V_1 \leftarrow V_1 \cup \{c1\}$; $V_2 \leftarrow V_2 \cup \{c2\}$
   (b) for each literal $s \in c$
      i. if s corresponds to a negation of a variable ($\overline{x}$) then $E \leftarrow E \cup \{(c1, s)\}$; $w(c1, s) \leftarrow (1, 10)$;

ii. else $E \leftarrow E \cup \{(c2, s)\}$; $w(c2, s) \leftarrow (10, 1)$;

(c) a copy of graph $\mathcal{Z}$ drawn in Fig. 3.1 whose nodes are named $\{c.a, c.b, c.x, c.y, c.r\}$ is a subgraph of G.

(d) $E \leftarrow E \cup \{(c1, c.a), (c2, c.x)\}$; $w(c1, c.a) \leftarrow (1, 1)$; $w(c2, c.x) \leftarrow (1, 1)$;

(e) $E \leftarrow E \cup \{(c.r, r)\}$;

(f) $w(c.r, r) \leftarrow (2, 2)$;

4. $V \leftarrow V_1 \cup V_2$; $\mathcal{P} \leftarrow (V_1, V_2)$

5. Return $(G, r, \mathcal{P}, w)$:

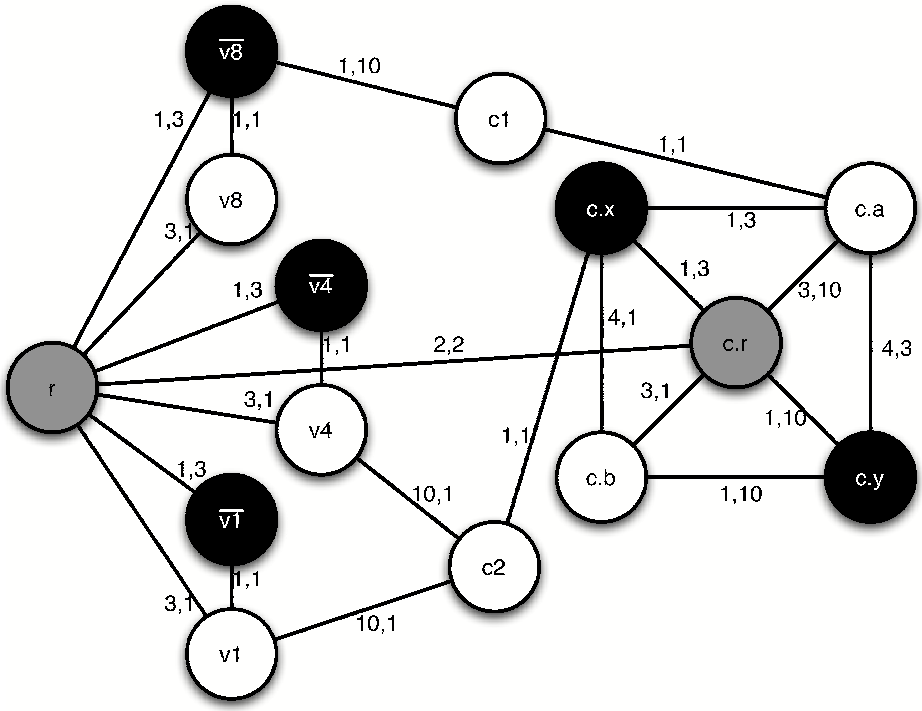

Figure 3.2: Example of the transformation with a clause $c = (\overline{v_1}, v_4, \overline{v_8})$

Fig 3.2 presents a partial example of this transformation. It is straightforward to verify that this transformation can be done in polynomial time in the size of the instance. We now prove that there exists a particular assignment t that satisfies all clauses of C if and only if there exists a stable tree T in G for metric w.

First, assume that there exists an assignment t that satisfies all clauses of C. We construct from t a tree T that is stable in G for metric w. For each variable $v \in U$, $(\overline{v}, v) \in T$. If $t(v) = true$, then $(r, v) \in T$ else $(t(v) = false)$, $(r, \overline{v}) \in T$. Nodes $\overline{v}$ and v are stable in T for w (since all other paths from v (or $\overline{v}$) to r are of weight strictly greater than 4). Moreover, for every $c \in C$, there exists at least one of those three literals v such that $t(v) = true$. Without loss of generality, we assume that v is the negation of a variable. T is constructed such that $(v, c2) \in T$ and $(c2, c.x) \in T$. Then, $(c.r, r) \in T$, $(c.r, c.y) \in T$ $(c.b, c.y) \in T$, $(c.a, c.r) \in T$. Moreover, $(c1, c.a)$ is in T if $c1$ has only one vertex in G and otherwise $(t, c1)$ is in T where t is a literal in c. Thus, T is a stable tree in V for metric w.

Second, assume that there exists a stable tree T in G for metric w. From this stable tree T, we construct an assignment t that satisfies all clauses of C. We consider an element $c = (v, v', v'') \in C$.

Let us first notice that the shortest path according to w_1 in G from $c.r$ to r is $(c.r, r)$. This permits to deduce that for every $c \in C$, $(c.r, r) \in T$. Assume now that the two paths from $c.a$ to r and from $c.x$ to r in T contain only nodes $c.x$, $c.y$, $c.b$, or $c.a$. Then, by construction of G, to cover nodes $c.a, c.b, c.x, c.y, c.r$, the shortest path tree T must use a subset of the edges of the complete graph induced by the set

of nodes $\{c.a, c.b, c.x, c.y, c.r\}$. This brings us back to the case of Fig 3.1 where there exists no stable shortest path tree. This contradicts the fact that the two paths from $c.a$ to r and from $c.x$ to r in T contain only nodes $c.x$, $c.y$, $c.b$, or $c.a$.

Thus, this implies that there exists a literal v' of clause c

1. such that the path from $c.a$ to r is in $\{c.a, c1, v, r\}$ if v is a negation of a variable z ($v = \overline{z}$).

2. or such that the path from $c.x$ to r is in $\{c.x, c2, v, r\}$ otherwise.

Without loss of generality, assume the path from $c.x$ to r in T includes v. This path does not include $\overline{v}$ (otherwise, $\{c.x, c1, v, \overline{v}, r\}$ would have a weight of 6 whereas $\{c.x, c.r, r\}$ would have a weight of 5). In this configuration, node $c.x$ is not stable in T. Thus, $(v, r) \in T$ and $(v, \overline{v}) \in T$. We now present the assignment $t : U \rightarrow \{true, false\}$, such that for each variable $v \in U$, $t(v) = true$ if and only if $(v, u) \in T$. From the previous remark, we deduce that t satisfies all clauses in C. □

We discuss the sufficient condition for detecting equilibrium for maximum flow tree in Chapter 4. The same condition holds for shortest path tree. We present a (weakly) stabilizing distributed algorithm to construct a rooted shortest path tree if a solution to the problem exists.

3.6 A Preliminary Attempt

Algorithms for constructing stabilizing rooted shortest path trees for only one class of processes are already known [8, 30]. We first demonstrate that a trivial

extension of it (such as each process selfishly picking an edge whose cost is minimal for its own color) for multiple classes of processes may fail to produce a stable tree.

Each process i maintains two variables: $\pi(i)$ and $\lambda(i)$ (commonly called the *parent* and the *level* or *distance* variables respectively). By definition, for the root node r, $\pi(r)$ is non-existent. Every other node picks a neighboring node as its parent (*i.e.* the domain of the $\pi(i)$ variable is $N(i)$). The level for each node i is $\lambda(i)$ that denotes its distance from the root. Clearly, $\lambda(r) = 0$. Also define

$$levelOK(i) \equiv \lambda(i) = \lambda(\pi(i)) + w(i, \pi(i))$$

A stabilizing solution for shortest path tree for a single class of processes is shown in Algorithm 3.1.

Algorithm 3.1 A stabilizing algorithm for rooted shortest path tree construction for a single class of processes.

- Fix level: $\neg\, levelOK(i) \longrightarrow \lambda(i) := \lambda(\pi(i)) + w(i, \pi(i))$

- Fix parent: $levelOK(i) \wedge \lambda(\pi(i)) + w(i, \pi(i)) \neq min\{\lambda(j) + w(i, j) : j \in N(i)\}$

  $\longrightarrow$ select $\pi(i)$: $\lambda(\pi(i)) + w(i, \pi(i)) = min\{\lambda(j) + w(i, j) : j \in N(i)\}$

Unfortunately, extensions of this approach to multiple classes of processes can be problematic. For example, assume that each process somehow figures out the shortest path route to the route for itself, and chooses its neighbor along its shortest path route as its parent. This may not lead to a shortest path tree as illustrated in

Figure 3.4. In Figure 3.4 there are two classes of processes: *black* and *white*. Each edge is labeled as (w, b) where w is the cost of that edge for a white node, and b is the cost of the same edge for a black node ($w = w_1(e)$, $b = w_2(e)$). Note that for each process, the minimum cost route to the designated root r now consists of two hops, the first hop leading to the non-root neighbor as parent. As a result, the first-hop links form a cycle instead of a tree, never connecting to the root. It is easy to see that a naive pure selfish strategy like that in Figure 3.3 also results in a cycle.

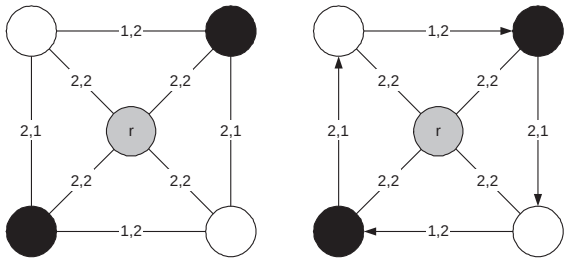

Figure 3.3: Pure selfish strategies may not yield a tree. Each edge is labeled as (w, b) where w is the cost of that edge for a white node, and b is the cost of the same edge for a black node ($w = w_1(e)$, $b = w_2(e)$). If each node picks its clockwise neighbor as its parent, then it results in a cycle.

The problem is non-trivial even if processes stick to a fixed strategy known to all. Strategy switch adds another level of complexity to it.

3.7 A Sufficient Condition for the Existence of a Stable Tree

This section summarizes our main results about selfish stabilization.

Lemma 1. *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V*

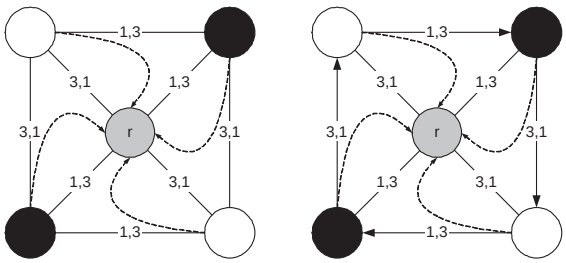

Figure 3.4: Pure selfish strategies may not yield a tree. Each edge is labeled as (w, b) where w is the cost of that edge for a white node, and b is the cost of the same edge for a black node ($w = w_1(e)$, $b = w_2(e)$). Based on this cost, the minimal cost route for each black or white node to the root node r consists of two hops as shown by the broken lines.

with 2 elements V_1 and V_2. Let w be a function on $E \to N^2$. Let v be a node of V_i (with $i \in \{1, 2\}$) such that there exists a shortest path between v and r using metric w_i that only contains nodes in V_i, then every stable tree T in G with metric w_i contains the shortest path between v and r.

Proof. Let $pcc = \{r, v_1, \ldots, v_\ell\}$ be a shortest path between v and r in metric w_i, solely composed of nodes of V_i. We prove this lemma by induction on parameter ℓ.

- If $\ell = 1$, then $w_i(pcc) = w_i(r, v_1)$. Let T be a stable tree such that $(r, v_1) \notin T$.

  As v_1 is stable in T, we have:

$$\forall z \in N_G(v_1), w_i(T_{v_1 \to r}) \leq w_i((v_1, z)) + w_i(T_{z \to r})$$

  As $r \in N_G(v_1)$, replacing z by r in the above inequality, we have:

$$w_i(T_{v_1 \to r}) \leq w_i(r, v_1) + w_i(T_{r \to r})$$

  now, $w_i(T_{r \to r})$ is 0, so we have:

$$w_i(T_{v_1 \to r}) \leq w_i(pcc)$$

(1)

Again,

$$w_i(T_{v_1 \to r}) > w_i(pcc)$$

(2)

This is because $T_{v_1 \to r}$ is not the shortest path from v_1 to r according to our assumption. So (1) and (2) give a contradiction. So the assumption (r, v_1) is not part of the stable tree was wrong.

if $(r, v1)$ is part of the stable tree, then T connects r to v_1 using the shortest path according to metric w_i. Here shortest path and direct edge are synonymous for the base case $\ell = 1$ as r is a neighbor of v_1, so we know there is a direct edge between them.

Thus, T connects r to v_1 using a shortest path according to metric w_i.

- Suppose now that the lemma is true for any j such that $0 < i < \ell$. Let T be a stable tree for metric w_i. By induction hypothesis, T connects u to $v_{\ell-1}$ through a shortest path according to metric w_i. As v_ℓ is stable in T, we have:

$$\forall z \in N_G(v_\ell), w_i(T_{v_\ell \to r}) \leq w_i((v_\ell, z)) + w_i(T_{z \to r})$$

As $v_{\ell-1} \in N_G(v_\ell)$, we have

$$w_i(T_{v_\ell \to r}) \leq w_i((v_{\ell-1}, v_\ell)) + w_i(T_{v_{\ell-1} \to r})$$

Thus, T connects r to v_ℓ through a shortest path according to metric w_i.

□

Theorem 2. *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with 2 elements V_1 and V_2, and such that V_1 contains exactly one node x. Let w be a function on $E \to N^2$. There always exists a stable tree T in G for metric w.*[2]

Proof. If $x = r$, we fall back in the known case of the construction of a shortest path spanning tree with metric w_2. In the sequel, we assume that $x \neq r$. The theorem is proved by induction on the degree d of x.

- If $d = 1$, every shortest path tree using metric w_2 is a stable tree since x has exactly one incident edge.
- Suppose now that the theorem is true for any graph where the degree of x is (strictly) lower than d. Let $z_1, \ldots, z_d$ be the neighbors of x. Let pcc_i be a shortest path between x and r including z_i according to metric w_2. For simplicity, we assume neighbors are sorted by increasing values of $w_2(pcc_i)$. Let us now consider the graph G_1 that is constructed in the following way:
  - $V(G_1) = V(G)$ and $E(G_1) = E(G) \setminus \{(x, z_1)\}$
  - $\forall e \in E(G_1)$, $w'(e) = w(e)$

  By induction hypothesis, there exists a stable tree T^1 in G_1 according to metric w'. Suppose that T^1 is not stable in G according to metric w. By definition of

[2]This theorem was proved with help from Johanne Cohen and Sébastien Tixeuil

w' and G_1, only x may not be stable. Thus, we have:

$$w_1((x, z_1)) + w_1(T^1_{z_1 \to r}) \leq w_1(T^1_{x \to r}) \tag{3.1}$$

Consider T that is a copy of T^1 except that the parent of x is z_1. Now, x is stable in T. We now prove that T is stable in G according to metric w. Let $Y = V \cap \{y : x \in T^1_{y \to r}\}$. First, every node in $V \backslash Y$ are stable. Now consider a node $y \in V_2 \cap Y$. Let t be the neighbor of y that belongs to the path $T^1_{y \to r}$. Now, y is stable in G_1 for metric w' (since $x \neq y$), so we get:

$$\forall s \in N(y) \backslash \{t\}, w_2(T^1_{y \to r}) \leq w_2(T^1_{s \to r}) + w_2((s, y)) \tag{3.2}$$

By definition, we have:

$$w_2(T_{y \to r}) = w_2(T^1_{y \to x}) + w_2((x, z_1)) + w_2(T^1_{z_1 \to r}) \tag{3.3}$$

The path $T^1_{z_1 \to r}$ connecting r to z_1 and belonging to T^1 is a shortest path according to metric w_2 since there exists a shortest path from r to w_1 according to metric w_2 whose nodes all belong to V_2 (see Lemma 1). Then, by definition of pcc_1 which is a shortest path from x to r through z_1, we have:

$$w_2(T^1_{z_1 \to r}) + w_2((x, z_1)) = w_2(pcc_1) \tag{3.4}$$

Combining Equations 3.3 and 3.4, we obtain:

$$w_2(T_{w \to r}) = w_2(T^1_{w \to x}) + w_2(pcc_1) \tag{3.5}$$

By definition, $\forall z_i \in N_G(x)$, we have $w_2(pcc_i) \geq w_2(pcc_1)$. Combining the previous remark and Equation 3.5 gives:

$$w_2(T_{y\to r}) = w_2(T^1_{y\to x}) + w_2(pcc_1) \leq w_2(T^1_{y\to x}) + w_2(pcc_i) \tag{3.6}$$

$$w_2(T_{y\to r}) \leq w_2(T^1_{y\to x}) + w_2(T^1_{x\to r}) \tag{3.7}$$

$$w_2(T_{y\to r}) \leq w_2(T^1_{y\to r}) \tag{3.8}$$

Combining Equations 3.2 and 3.8, we deduce that y is stable in tree T for metric w. Thus, T is stable in G for metric w. This concludes the proof.

□

Definition 11. *A* dispute wheel *[23] of size k correspond to:*

- *a graph G having a distinct subset $U(G) = \{u_0, u_1 \ldots, u_{k-1}\}$ of vertices such that there exist a unique path Q_j between u_j and u, and a unique path R_j between u_j and u_{j+1},*
- *a weighted function w and a partition of vertices such that, for any j, $0 \leq j \leq k-1$:*
  1. *u_j and u_{j+1} do not belong to the same element of partition,*
  2. *if $j > 0$, and if $u_j \in V_i$*

$$w_i(Q_j) < min(w_i(R_j), w_i(Q_{j+1}))$$

  3. *if $j = 0$,*

$$min(w_2(R_0), w_i(Q_1)) < w_2(Q_0)$$

$$w_2(Q_0) < min(w_2(R_0), w_i(R_1 \cdot Q_2))$$

with $u_0 \in V_2$.

Note that all subscripts must be interpreted modulo k.

Theorem 3. *Let* $G = (V, E)$ *be a graph and* r *a node of* V *not containing a* dispute wheel. *Let* $\mathcal{P}$ *be a partition of* V *with two elements* V_1 *and* V_2*, and such that* V_1 *contains two nodes* x_1 *and* x_2*. Let* w *be a function on* $E \rightarrow N^2$*. There always exists a stable shortest path tree* T *in* G *for metric* w*.*

Theorem 4. *If* G *contains no* dispute wheel*, there exists a stable shortest path tree.*

The proofs of these two theorems are in the same line as that of Theorem 11 and Theorem 12 in Chapter 4, which are proven for maximum flow tree.

Observation 1. *Stabilization may not be feasible if at the same time, more than one process make moves i.e., a distributed synchronous demon can play the role of an adversary and the configurations can repeat infinitely (Figure 3.5)*

Observation 2. *More than one equilibrium are possible with the same setting (Figure 3.7).*

Observation 3. *For a central demon, the processes can repeat the same sequence of configurations and stabilization may never be achieved (Figure 3.6).*

Observation 4. *For a randomized demon, the processes can not repeat the same sequence of configurations and stabilization is eventually achieved (Figure 3.8).*

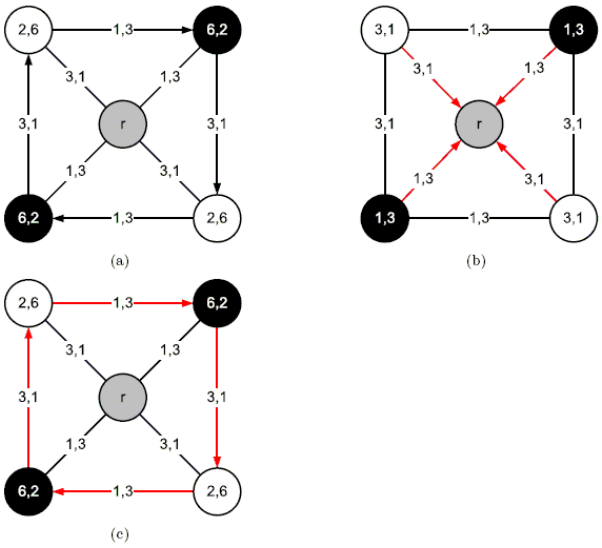

Figure 3.5: Example execution with the distributed synchronous demon. Configuration (a) and (c) are the same, so the system does not stabilize.

3.8 Stabilizing Algorithm

Our goal in this section is to design distributed algorithms that construct a stable tree T for a particular metric w. Since the system is distributed, nodes are only aware of their neighborhood.

We assume that each node i is aware of $N(i)$, the set of its neighbors (excluding i itself). Similarly, each node i is aware of the cost of each of its adjacent edges $e = (i, j) : j \in N(i)$. The cost of an edge e is a vector

$$w(e) = (w_1(e), w_2(e), w_3(e), \ldots, w_p(e))$$

where $w_k(e)$ denotes the cost of the edge e for a node of color k.

Also, i maintains two variables: $\pi(i)$ and $\lambda(i)$ (commonly called the *parent* and the *level* or *distance* variables). By definition, for the root node r, $\pi(r)$ is non-

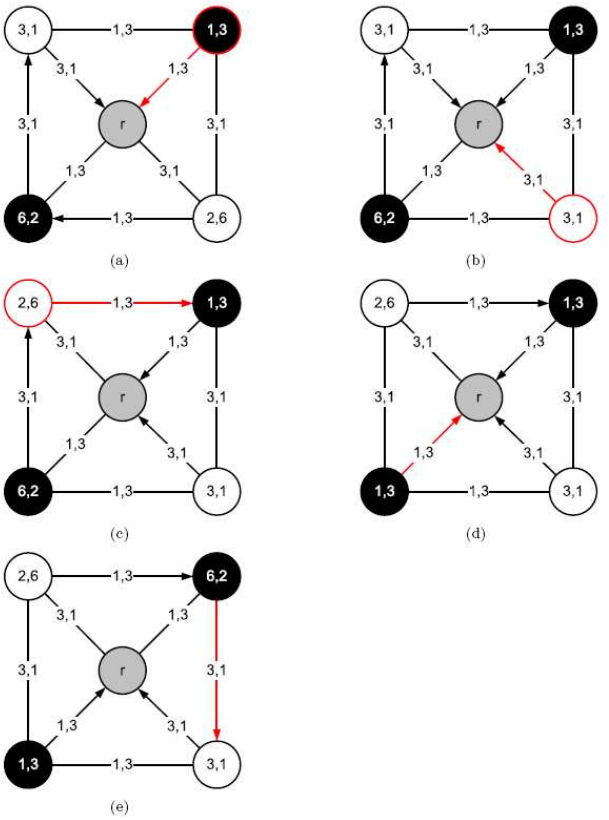

Figure 3.6: Example execution with the sequential (central) demon. Configurations (a) and (e) are symmetric, so repeating the sequence of four moves as shown leads to Configuration (a) everytime and the system will not stabilize.

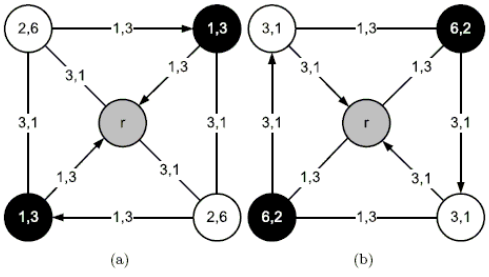

Figure 3.7: More than one equilibrium are possible with the same setting.

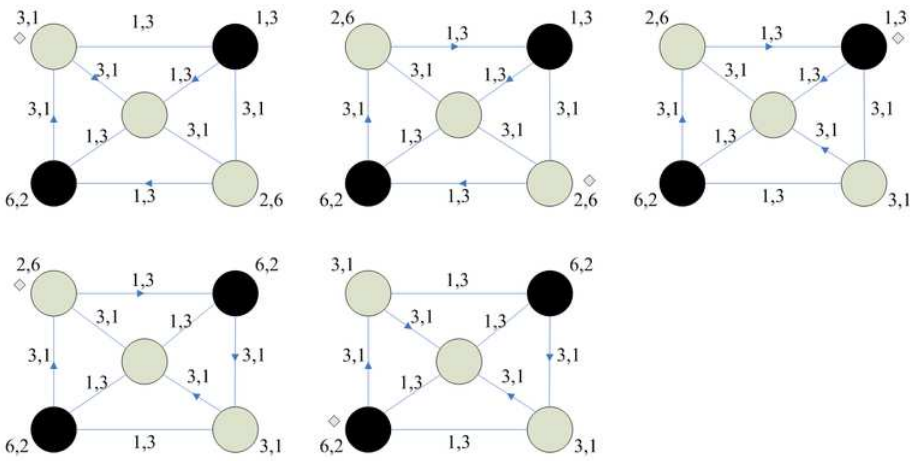

Figure 3.8: We start with Figure 3.6. The processes marked with diamonds make moves by changing their parent. Correspondingly the node costs are also adjusted. The diagram shows if we don't allow a sequence of processes to make moves repeatedly, i.e., a randomized scheduler can prevent repetition occurrence and stabilization can be achieved. The last configuration is stable and no node has any move, so the algorithm terminates with equilibrium being achieved.

existent. Every other node picks a neighboring node as its parent (*i.e.* the domain of the $\pi(i)$ variable is $N(i)$. The level for each node is a vector

$$\lambda(i) = (\lambda_1(i), \lambda_2(i), \lambda_3(i), \ldots, \lambda_p(i))$$

where $\lambda_k(i)$ denotes the distance for a node of the k^{th} color from node i to the root. By definition, $\lambda(r) = (0, 0, \ldots, 0)$.

In networks with selfish peers, the costs influencing decision-making are often of commercial nature, and are thus kept private to the participating nodes. To accommodate this feature in our framework, we assume that for each node, the value of each component of λ and the weights of the edges incident on it w will be stored in an encrypted form. All the same color nodes share a common secret key. Thus,

no node can access the component of the costs of a different color from a neighboring node or link to decide its course of action. However every node can securely extract the component of the variables corresponding to its own color. This authentication mechanism preserves fairness of the game and prevents possible foul play by deliberately tampering the variables of the nodes of opposing color. We will designate the encrypted version of a variable x by $\hat{x}$. For the sake of simplicity, we assume that [3]

$$\hat{x} + \hat{y} = \widehat{x + y}$$

We first propose a greedy algorithm for systems that contain nodes of two different colors 1 and 2 (*i.e.* $p = 2$): each node i of color k will select a parent $\pi(i)$ that minimizes $\lambda_k(i)$, regardless of what happens to $\lambda_j(i)$ $(j \neq k)$.

The proposed algorithm (algorithm 3.2, denoted afterward as Algorithm Greedy) has a two guarded actions. The root r does not execute any action. The other nodes execute local adjustment of their levels (to make it consistent with their parent's) and of their parent (in order to locally minimize the cost of the metric for the node color). Also, the level adjustment action has higher priority than the parent adjustment action.

3.9 Proof of Correctness

Theorem 5. *Let $G = (V, E)$ be a graph containing no dispute wheel and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements. Let w be a function on $E \rightarrow N^p$. If no node has enabled rules, the structure induced by the π variables of each node*

[3] Homomorphic encryptions like Paillier's scheme [40] satisfy this property.

Algorithm 3.2 A stabilizing algorithm for shortest path rooted tree construction for multiple classes of processes.

- Variable: $\lambda(i) \in N^p$, $\pi(i) \in N(i)$

- Macro: $levelOK(i) \equiv \widehat{\lambda(i)} = \widehat{\lambda(\pi(i))} + \widehat{w(i, \pi(i))}$

  $ParentOK(i)_{i \in V_k} \equiv \lambda_k(\pi(i)) + w_k(i, p(i)) = min\{\lambda_k(j) + w_k(i, j) : j \in N(i)\}$

  $Fixlevel(i) \equiv \widehat{\lambda(i)} := \widehat{\lambda(\pi(i))} + \widehat{w(i, \pi(i))}$

  $FixParent(i)_{i \in V_k} \equiv$ select $\pi(i)$: $\lambda_k(\pi(i)) + w_k(i, \pi(i)) = min\{\lambda_k(j) + w_k(i, j) :$

  $j \in N(i)\}$

- Actions: $i \neq r$

  - Fix level: $\neg\, levelOK(i) \longrightarrow Fixlevel(i)$;

  - Fix parent: $levelOK(i) \wedge \neg\, ParentOK(i) \longrightarrow FixParent(i)$.

executing Algorithm Greedy *induces a stable tree T rooted at r for metric w.*

Proof. Let us first prove that the induced structure is a tree. Due to the fact that the π variables are always pointing to a neighbor (for every node except r), the structure is either a tree or a tree with disconnected circuits. Assume for the purpose of contradiction that there exists at least one circuit. As w only has costs in N, all edge costs are (strictly) positive. As no level fixing rule is enabled, this implies that all levels are OK, and thus each level is strictly greater as the level of the parent. Due to the well founded property of the "lower than" relation on (positive) integers, the existence of such a circuit is impossible. Hence, the induced structure is a tree.

Now, we prove that the obtained tree T is stable for metric w. As no level fixing rules are enabled, this means that the λ variables are all properly computed from the local metric information w, that can not be corrupted. By induction on the distance (in number of hops) to the root r, this means that the λ variables contain the correct values for the cost of every path towards to root according to metric w (that is known from the w to every node). Now, the parent changing rules are also not enabled. This implies that the π variables are actually selecting the best neighbor according to the color k of node $i \in V_k$. This best neighbor being selected according to the metric w, we conclude that the tree T that is induced by the π variables is a stable tree for metric w. □

When there exists a single class of nodes in the system, the problem reduces to the problem of constructing a shortest path tree to a particular node r. There exist several self-stabilizing solutions to this problem (e.g.,[8]) that can be reused in

that particular case.

In the case there are exactly two classes of nodes, what we guarantee for Algorithm Greedy is *weak stabilization* [22]. Weak stabilization guarantees that starting from an arbitrary configuration, there exists *at least one computation* that leads the system to an equilibrium configuration.

Lemma 2. *Let $G = (V, E)$ be a graph containing no dispute wheel and r a node of V. Let $\mathcal{P}$ be a partition of V with 2 elements V_1 and V_2, and such that V_1 contains α nodes $x_1, x_2, \ldots x_\alpha$. Let w be a function on $E \rightarrow N^2$. Starting from a configuration where the π variables induce a single shortest path tree rooted at r, there exists an execution of algorithm Greedy that reaches a terminal configuration.*

Proof. Assume we start from a configuration where the π variables of all nodes induce a single shortest path tree. By the hypothesis that $p = 2$, we know there exists at least one stable tree T. Now consider the execution in which all nodes that have their level fixing rule enabled execute it, and that nodes that have their parent fixing rule don't execute it. Since the π variables induce a tree, only a fixed number of level fixing rules can be executed. If in this configuration, no node has a rule to execute, the configuration is terminal. Otherwise, this means that there exists at least one node that is willing to change its parent. When one node changes its parent, there exists an execution in which all nodes in its subtree update their level and no other parent fixing rule is executed. Using this scheme, we always guarantee that a single tree rooted at r induced by the π variables exists in the system.

Now we mimic the construction of the stable tree T in the proofs of Theorem 2

and 3, reasoning by induction on the number of nodes in V_1, and then by induction on the degree of each node in V_1. Anytime an edge is removed then replaced to move toward a stable tree, this correspond to a parent adjustment at some node in our algorithm. Since all level adjustments are made between any two parent adjustments, we conclude that the system eventually reaches a stable tree T considering the π variables. Now, when the tree induced by the π variables is stable, we consider the execution in which all levels are fixed. By the acyclic nature of the tree, there is a finite number of such moves. □

Lemma 3. *Let $G = (V, E)$ be a graph containing no dispute wheel and r a node of V. Let $\mathcal{P}$ be a partition of V with 2 elements V_1 and V_2, and such that V_1 contains α nodes $x_1, x_2, \ldots x_\alpha$. Let w be a function on $E \to N^2$. Starting from an arbitrary configuration, there exists an execution of Algorithm* Greedy *that reaches a configuration where the π variables induce a single shortest path tree rooted at r.*

Proof. In an arbitrary initial configuration, the π variables may induce either a tree, or a tree and a set of circuits. We consider the following execution: first all levels of nodes that are in the tree component and that need to be fixed are fixed by executing the level adjustment rule. Then, for every circuit, there exists at least one node that can adjust its level (due to the well boundedness of the "less than" relation on positive integers). Then, it turns out that for every cycle, there exists an execution that makes the level grow unbounded. We consider the execution that reaches a configuration where every node in a circuit has a level so high that it is greater than the highest level of any node in the tree component plus the weight of the edge for the node

color according to metric w. Also, there is at least one such circuit such that at least one node of the circuit is also a neighbor of one node in the tree component. Now, consider one node i that is a neighbor of a node in the tree component; this node may have to execute its level adjustment rule (in case it is needed), then it may execute (immediately after) its parent change rule. In this execution, after the *FixParent* move, all nodes belonging to the circuit containing i execute the *Fixlevel* move if needed. This execution leads to a configuration where the number of circuits is reduced by 1. The process can be repeated, and by induction on the number of initial circuits, the result is obtained. □

Theorem 6 (Weak Stabilization)**.** *Let $G = (V, E)$ be a graph containing no dispute wheel and r a node of V. Let $\mathcal{P}$ be a partition of V with 2 elements V_1 and V_2, and such that V_1 contains α nodes $x_1, x_2, \ldots x_\alpha$. Let w be a function on $E \rightarrow N^2$. Algorithm* Greedy *is weakly stabilizing for the stable tree rooted at r construction problem according to metric w.*

Proof. Starting from an arbitrary initial configuration, there exists an execution that reaches a configuration where a tree is induced by the π variables (Lemma 3). From such a configuration, there exists an execution that reaches a terminal configuration (Lemma 2). Every terminal configuration denotes a stable tree according to metric w. □

Of course, our (weakly) stabilizing solution does not preclude the existence of (strongly) stabilizing solutions, but proving the existence of such solution is an open

question.

3.10 Alternative Strategies

If there exists a set of strategies (synonymous with algorithms) with the property that no process can lower its cost by changing its strategy while the other processes keep their strategies unchanged, then that set of strategies and the corresponding costs constitute the *Nash Equilibrium*.

To prove that the stable configuration reflects a Nash equilibrium, we need to consider various strategies that can be adopted by the processes to lower their costs. The Greedy algorithm we proposed uses a *greedy strategy*, (call it *Strategy A*) but it is, by no means, the only possible strategy. Let us examine a second strategy for cost minimization by the individual processes. It is an *altruistic strategy*: each node picks a parent that lowers the communication cost of the nodes of the opposite color (call it *Strategy B*). As a result, processes in V_1 will help lower the cost of the processes in V_2, and vice versa. To implement Strategy B, we modify the definition of $ParentOK$ and $FixParent$ as follows (considering $\overline{\lambda}(i)$ denotes the level for the other color than i, and that $\overline{w}(i,k)$ denotes the weight for the other color than i) [4]:

Once these are appropriately defined, the main algorithm remains unchanged. Using the same line of arguments, we can show that this algorithm also stabilizes the system, but to a different configuration. This leads to the following observation:

---

[4]This apparently weakens the encryption mechanism since it requires $x > y \Rightarrow \hat{x} > \hat{y}$. However, using the altruistic protocol, processes in V_1 lower their cost by helping the processes in V_2 and vice versa, and this is more conducive to building a trust relationship. So we will disregard the encryption symbol.

Algorithm 3.3 Alternative algorithm

- Variables: $\lambda(i) \in N^p$, $\pi(i) \in N(i)$
- Macro: $ParentOK(i) \equiv \pi(i) = j : \overline{\lambda}(j) + \overline{w}(i,j) = min\{\overline{\lambda}(k) + \overline{w}(i,k) : k \in N(i)\}$

  $FixParent(i) \equiv$ select $\pi(i) := j : \overline{\lambda}(j) + \overline{w}(i,j) = min\{\overline{\lambda}(k) + \overline{w}(i,k) : k \in N(i)\}$

  $levelOK(i) \equiv \overline{\lambda}(i) = \overline{\lambda}(\pi(i)) + \overline{w}(i, \pi(i))$

  $Fixlevel(i) \equiv \overline{\lambda}(i) := \overline{\lambda}(\pi(i)) + \overline{w}(i, \pi(i))$
- Actions: $i \neq r$
  - Fix level: $\neg\, levelOK(i) \longrightarrow Fixlevel(i)$;
  - Fix parent: $levelOK(i) \wedge \neg\, ParentOK(i) \longrightarrow FixParent(i)$.

Observation 5. *Using Strategy B, the system of processes (weakly) stabilizes to an equilibrium configuration, and the edges connecting the processes with their parents form a shortest path tree.*

The observation trivially follows from Theorem 6 if we swap the costs of the nodes in V_1 and V_2 for each edge.

The Cost of Equilibrium. A natural component of such an exercise is to analyze the quality of the equilibrium configuration: How bad is the cost of this configuration in comparison with the "optimal" configuration? For the nodes of a given color, define the cost of a configuration as the sum of weights of all the tree edges for that color. Define the *optimal* cost as the cost of the tree when all nodes are of the same color.

The issue is: By what extent will it increase if some of the nodes belong to a different color? Here is an upper bound. Let $e_{max} = max\{w(e), \overline{w}(e) : e \in E\}$ and $e_{min} = min\{w(e), \overline{w}(e) : e \in E\}$. Then the following theorem holds.

Theorem 7. *For any set of processes of a given color, the ratio of the cost of the equilibrium configuration to the cost of the optimal configuration is bounded from above by* $\frac{e_{max}}{e_{min}}$.

Proof. A tree with N nodes has $(N-1)$ edges, so the cost of the optimal configuration has a lower bound of $(N-1).e_{min}$. To determine the maximum possible weight of the tree in an equilibrium configuration under any of the algorithms A or B (or a mix of the two), think of an adversary that can switch the color of zero or more processes so that each node chooses the edge with largest weight as its link to its parent node. The cost of the resulting configuration is bounded from above by $(N-1).e_{max}$. The ratio of the two costs will not exceed $\frac{e_{max}}{e_{min}}$ □

This is a loose upper bound. In general, when the number of processes in V_1 is much larger than the number of processes in V_2, Strategy A will lead to a lower cost for the processes in V_1, and Strategy B will lead to a lower cost for the processes in V_2. This is quite intuitive, since in Strategy A, each step by the majority (*i.e.* in V_1) processes helps lower their own cost at the expense of the competitors' cost, whereas in Strategy B, each step by the majority processes lowers the cost of the competitors at the expense of their own cost.

Observation 6. *The cost of the processes in* V_1 *(resp.* V_2*) will be minimum when*

they use Strategy A while the processes in V_2 (resp. V_1) use strategy B.

Viewed from the perspective of the processes in V_1, the validity of the above observation is based on the fact that that every node picks the best edge for the processes in V_1, so the algorithm reduces to the classical stabilizing shortest path algorithm (*e.g.*, [8]) for the nodes in V_1.

Theorem 8. *For a given graph $G = (V, E)$ with a given composition of the processes in V, and the set of strategies (A, B) the equilibrium configuration is unique, and it reflects the Nash equilibrium.*

Proof. Assume that using whatever strategy the processes choose, the system of processes stabilizes to some configuration that determines the payoffs for the processes in the V_1 and V_2 sets. Now consider three different cases:

1. Assume that all processes use Strategy B. Observe that one or more processes of a certain group will switch to Strategy A, since this will lower their cost (Theorem 3). However the other group might apprehend this, they will also switch from Strategy B to Strategy A.

2. Assume that the processes in V_1 use Strategy B, while the processes in V_2 use Strategy A. However, altruism does not pay off unless everyone is altruistic. Since the processes in V_1 do not know what strategy the processes in V_2 are using, they will switch to Strategy A, and their cost will go down.

3. Assume that all processes use Strategy A. Now no process will be motivated to switch to Strategy B, since such a switch will imply lowering the cost of the

other group even if it increases the cost of its own group. Thus this is a stable configuration.

Thus, regardless of the initial strategies chosen by the processes in V_1 and V_2, all processes will eventually switch to Strategy A, and regardless of the initial values of L and p, the system will stabilize in a bounded number of steps. Furthermore, since no process can unilaterally lower its cost by switching to a different strategy, the stable configuration will reflect a Nash equilibrium. □

Note. Both strategies (A and B) can be further optimized as follows. Consider A first. There may be cases in which a node $i \in V_1$ finds multiple neighbors j satisfying the condition of being a "best parent". Instead of arbitrarily choosing one such node, i will choose a $\pi(i) = j$ for which the cost of the opposite color component of $\lambda(i)$ is the lowest. A similar step can be taken by the nodes in V_2 too. The interesting aspect of this exercise is that not only does the network state stabilize to a desirable configuration, but the strategies stabilize too, in as much as regardless of the starting strategies, all processes end up using the same final strategy. This does not rule out the invention of new strategies beyond what has been considered for this exercise.

3.11 Discussions

In [24, 23], Griffin et al. considered the stable path problem, where every individual node has preferences on every path toward the destination. In [24], the authors showed that the problem of deciding upon the existence of routing equilibrium is NP-complete or NP-hard depending on the problem statement. In [23], the

same authors presented a sufficient condition (dispute wheels) on path preferences so that the equilibrium exists. In our case, we also find the sufficient condition for the existence of a stable tree, although the problem is strictly weaker, as "groups" have preferences on edges toward a destination. This could have led to complexity weakenings, but in [10], we showed that it is not the case, as bad networks from [24] could be translated to our constraints.

CHAPTER 4
SELFISH STABILIZATION IN MAXIMUM FLOW TREE

4.1 Introduction

This chapter formulates the problem of finding a *maximum flow tree* in the *selfish stabilization* environment and proposes possible solutions provided equilibrium exists in the system. Given a graph $G = (V, E)$, assume that there are p disjoint subsets (or colors) of nodes V_1, V_2,...,V_p, so that $V = \cup_{i=1}^{p} V_i$. We convert G into a multi-weighted graph by defining a cost function w of $E \rightarrow N^p$, where N is the set of positive integers. For every $i \in 1 \ldots p$, the function $w_i : E \rightarrow N$ denotes set V_i's cost of using edge e (the *capacity* value). Starting from an arbitrary initial configuration, the p different colors of nodes cooperate with one another to form a *flow tree* at a designated node (called the root), and at the same time compete against each other to maximize their *flow* value to the root, which acts as a sink.

The motivation of our problem is drawn from the real world scenario of bandwidth allocation [21]. In a practical scenario, different processes sometimes compete among themselves to utilize the bandwidth of an available channel. Here we try to model that as a variation of the *maximum flow* problem. Like traditional *maximum flow* problem, each edge is assigned a *capacity* value, but because of the presence of different subsets of processes, the *capacity* of each edge (channel) is shared among the different subsets (based on some payment scheme, nodes may buy part or whole of the *capacity*).

Flow of each vertex in a *flow tree* is computed by applying the *min* function to the *flow* of its parent in the tree and the *capacity* of the edge to its parent in the tree. In a *maximum flow tree*, for every vertex, its path along the tree has the maximum possible *flow* to the root [21], regardless of the color of the vertices. Note that such configurations may not be unique, or convergence to such a configuration may not be feasible. Nevertheless, each process will *try* to construct such a tree so that it can push as much flow as possible to the root using the available path. The formulation in [21] does not allow a flow to fork into multiple paths and then join later, since that will violate the underlying tree topology. We stick to these assumptions.

4.2 Contributions

We present examples of the existence of non-equilibrium configuration, where no matter how the scheduler chooses its sequence of actions, the system does not stabilize and analyze the sufficient condition for the existence of a stable tree, i.e., an equilibrium, in the system. We also consider the role of both central scheduler and synchronous scheduler in the context of *selfish stabilization*. We examine conditions under which, starting from an arbitrary initial configuration, different classes of processes can stabilize to an equilibrium configuration after which no process can unilaterally increase its flow value to the root.

4.3 Model and Notation

We follow the general model of selfish stabilization we introduced in Chapter 2. In addition to that, for the *maximum flow tree* problem, we consider the following

specific notations:

For the *maximum flow tree* problem, the cost of this path for a node belonging to a particular subset V_i is given by

$$w_i(T_{v\to u}) = min_{e\in T_{v\to u}} w_i(e)$$

where the *min* function returns the minimum cost (capacity) edge belonging to the path from u to v. The *flow* of each vertex in a *flow tree* is computed by applying the *min* function to the *flow* of its parent in the tree and the *capacity* of the edge to its parent in the tree. In a *maximum flow tree*, for every vertex, its path along the tree has the maximum possible *flow* of any path to the root.

In this chapter, we study the problem of constructing a tree T so that for each node v belonging to v_i, the flow that can be pushed from v to the root r (called cost(v)) is maximized. This cost is equal to $cost(v) = w_i(T_{v\to r})$.

This is the private goal for every node v. A node v is stable in a tree T if it has no incentive to chose another neighbor $z \in N_G(v)$.

Stable Node Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \to N^p$. Let T be a tree of G. The node $v \in V_i$ is *stable in T for metric w* if and only if it satisfies the following condition for the *maximum flow tree*:

$\forall z \in N_G(v)$,

$min_{e\in T_{v\to r}} w_i(e) \geq min(w_i((v, z)), min_{e\in T_{z\to r}} w_i(e))$

In other words,

$$\forall z \in N_G(v), w_i(T_{v\to r}) \geq min(w_i((v, z)), w_i(T_{z\to r}))$$

Similarly, a tree T is stable if every node in T is stable.

Stable Tree Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \rightarrow N^p$. Let T be a tree of G. The tree T is *stable for metric* w if and only if for every node $v \in V$, v is stable in T for metric w.

Our goal is to devise a stabilizing mechanism for the construction of a stable tree given a particular metric – the computation should lead to a tree configuration so that the conflicting private goals of cost optimization do not interfere with the common goal of tree formation.

4.4 Non-existence of Equilibrium: An Example

We show an example of non-equilibrium for the *maximum flow tree* problem when processes are selfish. Fig 4.1 shows an example where there exists no stable tree for the *maximal flow path*, i.e., there exists no equilibrium with a 2-partition of the nodes of a graph with 5 nodes (called graph $\mathcal{Z}$). The set of nodes V is split into white nodes and black nodes: $white = \{r, v_1, v_3\}$, $black = \{v_2, v_4\}$. The metric w is described in Fig 4.1. By contradiction it is easy to verify that there exists no stable tree here: assume there exists a stable tree T. Now, if edge (v_2, r) is in T, then edge (v_3, v_2) should also be in T (otherwise T would not be stable because the maximum flow path between v_3 and r for metric w_1 is (v_3, v_2, r)). Now edge (v_4, v_3) is in T because path (v_4, v_3, v_2, r) is a maximum flow path between node v_4 and r for metric w_2. Now consider node v_1. Edges (v_1, v_2) and (v_1, v_4) cannot be in T. So, edge

(v_1, r) must be in T. This contradicts the stable property of T because at node v_2, we would have $min_{e \in T_{v_2 \to r}} w_2(e) = 7 \leq min(w_2((v_2, v_1)), min_{e \in T_{v_1 \to r}} w_i(e) = 9$. The same argument can be applied for edges (v_3, r), (v_1, r), and (v_4, r) by assuming one of them is in T. So there exists no stable tree for this instance of w. Of course, this example can be generalized to an arbitrary $p \geq 2$.

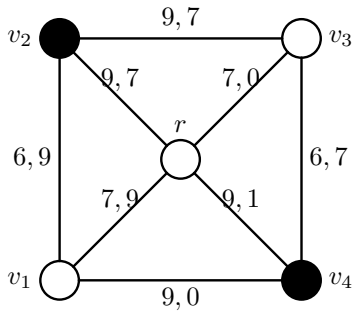

Figure 4.1: The graph $\mathcal{Z}$ with 5 nodes. This is an example with two colors where there does not exist an equilibrium in the system. Each edge is weighted (white color, black color). By contradiction, one can show that there is no stable tree $\mathcal{Z}$

Clearly, there is no point in searching for a stabilizing solution, because no such solution can be found.

4.5 The Decision Problem: Does Equilibrium Exist?

We now discuss the existence of an equilibrium (*i.e.* a stable maximum flow tree T for a particular metric w) in relation to the number of conflicting interests (*i.e.* the number p in the partition of nodes of V into p sets $V_1, V_2, \ldots, V_p$). The special case of $p = 1$ is the usual problem of the maximum flow tree with non zero edge costs, so an equilibrium always exists. We first prove that when $p \geq 2$, determining whether

there exists an equilibrium for a particular metric w is an NP-complete problem.

Theorem 9. *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with $p \geq 2$ elements $V = \cup_{i=1}^{p} V_i$. Let w be a function on $E \rightarrow N^p$. The problem of deciding the existence of a stable maximum flow tree T for metric w is NP-complete.*[1]

Proof. It is easy to show that this problem is in NP. Indeed, given a particular tree T, checking whether T is stable for metric w can be done in polynomial time by checking if every node is stable in T for metric w.

Then, we construct a polynomial transformation (almost similar to transformation described in [24]) with the NP-complete problem 3-SAT defined as follows:

Instance: Collection $C = \{c_1, c_2, \ldots c_m\}$ of clauses on a finite set U of variables such that $|c_i| = 3$ for $1 \leq i \leq m$.

Question: Is there a truth assignment for U that satisfies all the clauses in C?

We now present the transformation $\mathcal{R} : (C, U) \rightarrow (G, r, \mathcal{P}, w)$:

1. $V_1 \leftarrow \{r\}$; $V_2 \leftarrow \emptyset$; $E \leftarrow \emptyset$;
2. for each variable $z \in U$ do:
   (a) $V \leftarrow V \cup \{z, \overline{z}\}$;$V_1 \leftarrow V_1 \cup \{z\}$;$V_2 \leftarrow V_2 \cup \{\overline{z}\}$;
   (b) $E \leftarrow E \cup \{(z, r), (\overline{z}, r), (z, \overline{z})\}$;
   (c) $w(z, r) \leftarrow (6, 11)$; $w(\overline{z}, r) \leftarrow (11, 6)$; $w(z, \overline{z}) \leftarrow (11, 11)$;

[1]This theorem was proved with help from Johanne Cohen and Sébastien Tixeuil

3. for each element $c \in C$ do:

   (a) a copy of graph $\mathcal{Z}$ drawn in Fig. 4.1 whose nodes are named $\{c.v_1, c.v_2, c.v_3, c.v_4, c.r\}$ is a subgraph of G.

   (b) $w(c.r, r) \leftarrow (20, 20)$;

   (c) for each literal $s \in c$

      i. if s corresponds to a negation of a variable ($\overline{x}$) then $E \leftarrow E \cup \{(c.v_1, s)\}$; $w(c.v_1, s) \leftarrow (10, 0)$;

      ii. else $E \leftarrow E \cup \{(c.v_2, s)\}$; $w(c.v_2, s) \leftarrow (0, 10)$;

   (d) $E \leftarrow E \cup \{(c.r, r)\}$;

4. $V \leftarrow V_1 \cup V_2$; $\mathcal{P} \leftarrow (V_1, V_2)$

5. Return $(G, r, \mathcal{P}, w)$:

It is straightforward to verify that this transformation can be done in polynomial time in the size of the instance. We now prove that there exists a particular assignment t that satisfies all clauses of C if and only if there exists a stable maximum flow tree T in G for metric w.

First, assume that there exists an assignment t that satisfies all clauses of C. We construct from t a tree T that is stable in G for metric w. For each variable $z \in U$, $(\overline{z}, v) \in T$. If $t(z) = true$, then $(r, z) \in T$ else $(t(z) = false)$, $(r, \overline{z}) \in T$. Nodes $\overline{z}$ and z are stable in T for w (since all other paths from z (or $\overline{z}$) to r contain at most one edge having weight 0). Moreover, for every $c \in C$, there exists at least one of those

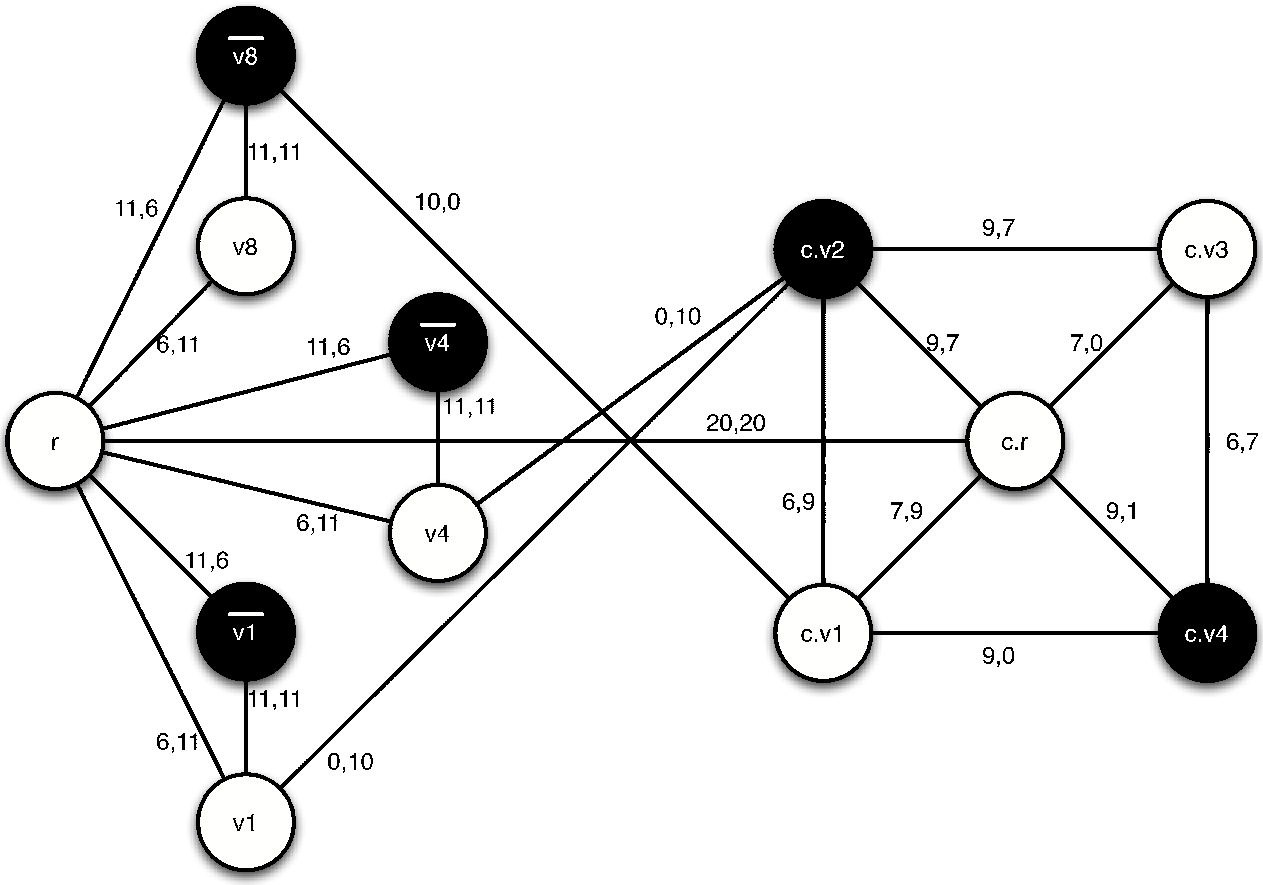

Figure 4.2: Example of the transformation with a clause $c = (\overline{v_1}, v_4, \overline{v_8})$

three literals s such that $t(s) = true$. Without loss of generality, we assume that s is the negation of a variable. T is constructed such that

- $(c.r, r) \in T$, (since all other paths from $c.r$ to r contain at most one edge having weight less than 20)
- $(c.v_1, s) \in T$ (because the cost of path $T_{c.v_1 \to r}$ is 10 and all other paths from $c.v_1$ to r contain at most one edge having weight less than 10)

Moreover, if c contains an positive literal s' such that $t(s') = true$, then $(c.v_2, s') \in T$

using the same argument for the fact that $(c.v_1, s) \in T$. We can deduce that $(c.v_3, c.v_2) \notin T$ and $(c.v_4, c.v_1) \notin T$ (because paths $\{c.v_4, c.v_1, s, r\}$ and $\{c.v_3, c.v_2, s', r\}$ contain an edge having weight less than 0). So $(c.v_4, c.r) \in T$ and $(c.v_3, c.r) \in T$. Otherwise, (if c does not contain an positive literal s' such that $t(s') = true$), $(c.v_2, c.r) \in T$ so that $c.v_2$ is stable. So $(c.v_3, c.v_2) \in T$ and $(c.v_3, c.v_4) \in T$.

Second, assume that there exists a stable tree T in G for metric w. From this stable tree T, we construct an assignment t that satisfies all clauses of C. We consider an element $c = (s, s', s'') \in C$.

Let us first notice that the maximun flow path according to w_1 in G from $c.r$ to r is $(c.r, r)$. This permits to deduce that for every $c \in C$, $(c.r, r) \in T$. Assume now that the two paths from $c.v_1$ to r and from $c.v_2$ to r in T contain only nodes $c.v_1$, $c.v_2$, $c.v_3$, or $c.v_4$. Then, by construction of G, to cover nodes $c.v_1, c.v_2, c.v_3, c.v_4, c.r$, the spanning tree T must use a subset of the edges of the complete graph induced by the set of nodes $\{c.v_1, c.v_2, c.v_3, c.v_4, c.r\}$. This brings us back to the case of Fig 4.1 where there exists no stable spanning tree. This contradicts the fact that the two paths from $c.v_1$ to r and from $c.v_2$ to r in T contain only nodes $c.v_1$, $c.v_2$, $c.v_3$, or $c.v_4$.

Thus, this implies that there exists a literal s of clause c

1. such that the path from $c.v_1$ to r is in $\{c.v_1, s, r\}$ if s is a negation of a variable z ($s = \overline{z}$).

2. or such that the path from $c.v_2$ to r is in $\{c.v_2, s, r\}$ otherwise.

Without loss of generality, assume the path from $c.v_1$ to r in T includes s. This

path does not include $\overline{s}$ (otherwise, $\{c.v_1, s, \overline{s}, r\}$ would have a weight of 6 whereas $\{c.v_1, c.r, r\}$ would have a weight of 7). In this configuration, node $c.v_1$ is not stable in T. Thus, $(s, r) \in T$ and $(s, \overline{s}) \in T$. We now present the assignment $t : U \rightarrow \{true, false\}$, such that for each variable $s \in U$, $t(s) = true$ if and only if $(s, r) \in T$. From the previous remark, we deduce that t satisfies all clauses in C. □

4.6 A Sufficient Condition for Stable Tree

Theorem 10. *Let $G = (V, E)$ be a graph and r be a node of V. Let $\mathcal{P}$ be a partition of V with 2 elements V_1 and V_2, and such that V_1 contains exactly one node x. Let w be a function on $E \rightarrow N^2$. There always exists a stable tree T in G for metric w.*

Proof. If $x = r$, we fall back to the known case of the construction of a maximum flow spanning tree with metric w_2. In the sequel, we assume that $x \neq r$. The theorem is proved by induction on the degree d of x.

If $d = 1$, every maximum flow tree using metric w_2 is a stable tree since x has exactly one incident edge.

Suppose now that the theorem is true for any graph where the degree of x is (strictly) lower than d. Let $z_1, \ldots, z_d$ be the neighbors of x. Let pcc_i be a maximum flow path between x and r including z_i according to metric w_2. For simplicity, we assume neighbors are sorted by decreasing values of $w_2(pcc_i)$. Let us now consider the graph G_1 that is constructed in the following way:

- $V(G_1) = V(G)$, $E(G_1) = E(G) \setminus \{(x, z_1)\}$
- $\forall e \in E(G_1)$, $w'(e) = w(e)$

By induction hypothesis, there exists a stable tree T^1 in G_1 according to metric w'. Suppose that T^1 is not stable in G according to metric w. By definition of w' and G_1, only x may not be stable. Thus, we have:

$$min(w_1((x, z_1)), w_1(T^1_{z_1 \to r})) > w_1(T^1_{x \to r})$$

Consider T that is a copy of T^1 except that the parent of x is z_1. Now, x is stable in T. We now prove that T is stable in G according to metric w. Let $Y = V \cap \{y : x \in T^1_{y \to r}\}$. First, every node in $V \backslash Y$ is stable. Now consider a node $y \in V_2 \cap Y$. Let t be the neighbor of y that belongs to the path $T^1_{y \to r}$. Now, y is stable in G_1 for metric w' (since $x \neq y$), so we get:

$$\forall s \in \Gamma(y) \backslash \{t\}, w_2(T^1_{y \to r}) \geq min(w_2(T^1_{s \to r}), w_2((s, y))) \tag{4.1}$$

By definition, we have:

$$w_2(T_{y \to r}) = min(w_2(T^1_{y \to x}), w_2((x, z_1)), w_2(T^1_{z_1 \to r})) \tag{4.2}$$

The path $T^1_{z_1 \to r}$ connecting r to z_1 and belonging to T^1 is a maximal flow path according to metric w_2 since there exists a maximal flow path from r to w_1 according to metric w_2 whose nodes all belong to V_2. Then, by definition of pcc_1 which is a maximal flow path from x to r through z_1, we have:

$$min(w_2(T^1_{z_1 \to r}), w_2((x, z_1))) = w_2(pcc_1) \tag{4.3}$$

Combining Equations 4.2 and 4.3, we obtain:

$$w_2(T_{w \to r}) = min(w_2(T^1_{w \to x}), w_2(pcc_1)) \tag{4.4}$$

By definition, $\forall z_i \in \Gamma_G(x)$, we have $w_2(pcc_i) \leq w_2(pcc_1)$. Combining the previous remark and Equation 4.4 gives:

$$w_2(T_{y\to r}) \geq min(w_2(T^1_{y\to x}), w_2(pcc_i)) \tag{4.5}$$

$$w_2(T_{y\to r}) \geq min(w_2(T^1_{y\to x}), w_2(T^1_{x\to r})) \tag{4.6}$$

$$w_2(T_{y\to r}) \geq w_2(T^1_{y\to r}) \tag{4.7}$$

Combining Equations 4.1 and 4.7, we deduce that y is stable in tree T for metric w. Thus, T is stable in G for metric w. This concludes the proof. □

Theorem 11. *Let $G = (V, E)$ be a graph and r a node of V not containing a* dispute wheel. *Let $\mathcal{P}$ be a partition of V with two elements V_1 and V_2, and such that V_1 contains two nodes x_1 and x_2. Let w be a function on $E \to N^2$. There always exists a stable maximum flow spanning tree T in G for metric w.*[2]

Proof. Without loss of generality, we assume that $r \in V_2$. We prove the theorem by constructing a sequence of spanning trees. First, we introduce some notations :

- $br(z, T) = max_{x\in\Gamma_G(z)}(min(w_i(x, z), w_i(T_{x\to r})))$.
- $\mathcal{BR}(z, T) = argmax_{x\in\Gamma_G(z)} min(w_i(x, z), w_i(T_{x\to r}))$.

where z is a vertex belonging to V_i and T is a spanning tree of G.

Node x_1 has d neighbors in G denoted by $z^1_1, \ldots, z^1_d$. We prove by induction on d that there exists a stable tree T in G for metric w.

[2]This theorem was proved with help from Johanne Cohen and Sébastien Tixeuil

If $d = 1$, it is sufficient to compute the stable tree T in $G \subset \{x_1\}$ using metric w (there exists such a tree by Theorem 10). Then, the edge incident to x_1 is added to T. Such a tree is stable in G for metric w.

Suppose now that the theorem is true for any graph where x_1 has degree (strictly) less than d. Let $z_1^1, \ldots, z_d^1$ be the neighbors of x_1. Consider now the graph G_1 that is constructed in the following way:

- $V(G_1) = V(G)$ and $E(G_1) = E(G) \setminus \{(x_1, z_1^1)\}$,
- $\forall e \in E(G_1)$, $w'(e) = w(e)$

By induction hypothesis, there exists a stable tree $T^{(1)}$ in G_1 according to metric w'.

We prove the induction step by contradiction : all spanning trees T are not stable in G according to metric w. So by assumption, $T^{(1)}$ is not stable in G according to metric w. So we get:

$$min(w_1((x_1, z_1^1)), w_1(T^{(1)}_{z_1^1 \to r})) \geq w_1(T^{(1)}_{x_1 \to r}) \tag{4.8}$$

Consider $T^{'}$, that is a copy of $T^{(1)}$ except that the parent of x_1 is z_1^1. Now, x_1 is stable in $T^{'}$.

$$min(w_1(T^{(1)}_{z \to r}), w_1(z, x_1)) > w_1(T^{(1)}_{x_1 \to r}) \tag{4.9}$$

Since vertices x_1 and x_2 belong to V_1, x_2 is also stable. By hypothesis, since no tree is stable, some vertices in V_2 are not stable: only neighbors of x_1 may be not stable in $T^{'}$. And, vertex y in V_2 which is not stable in $T^{'}$ is such that $x_1 \in T^{(1)}_{y \to r}$ and hence $x_1 \in T^{'}_{y \to r}$.

We transform tree $T^{'}$ into tree $T^{(2)}$ such that

- all vertices in V_2 will be stable,
- x_2 has the same parent in $T^{(2)}$ as in $T^{'}$

Let $\mathcal{U} \subseteq \Gamma_G(x_1)$ be a set of unstable vertices in $T^{'}$. We consider a vertex v such that $\forall y \in \mathcal{U}$, $br(v, T^{'}) \geq br(y, T^{'})$. Tree $T^{'}$ is modified so that a vertex of $\mathcal{BR}(v, T^{'})$ is a father of v. In this new tree, only neighbors of x_1 or of v belonging to V_2 can not be stable. This modification can be performed with an other vertex not stable, and so one until all vertices in V_2 are stable.

By hypothesis, tree $T^{(2)}$ is not stable, hence the following properties are satisfied:

1 only x_2 is not stable,

2 $T^{'}_{x_1 \to r} = T^{(2)}_{x_1 \to r}$ (because all vertices in V_2 are stable in $T^{(1)}_{x_1 \to r}$.

The fact that x_2 is not stable in tree $T^{(2)}$ is due to the following observations:

Case 1: $w_2(T^{'}_{x_1 \to r}) > w_2(T^{(1)}_{x_1 \to r})$. So it implies that there exists vertex y_1 in V_2 such that $T^{(2)}_{y \to r} \neq T^{'}_{y \to r}$ and such that there exists $z \in BR(x_2, T^{(2)})$ with $y_1 \in T^{(2)}_{z \to r}$.

Case 2: $w_2(T^{'}_{x_1 \to r}) < w_2(T^{(1)}_{x_1 \to r})$. So it implies there exists vertex y_1 in V_2 such that $T^{(2)}_{y_1 \to r} \neq T^{'}_{y_1 \to r}$ and such that $y_1 \in T^{'}_{x_2 \to r}$.

Case 1: Assume that $y_1 \in V_2$ is such that $T^{(2)}_{y_1 \to r} \neq T^{(')}_{y_1 \to r}$ and such that there exists $z \in BR(x_2, T^{(2)})$ with $y_1 \in T^{(2)}_{z \to r}$.

From Property **(2)** of tree $T^{(2)}$ and from the definition of y_1, we get

$$min(w_2(T^{(1)}_{y_1 \to x_1}), w_2(T^{(1)}_{x_1 \to r})) \geq w_2(T^{(2)}_{y_1 \to r}) \tag{4.10}$$

$$min(w_1(T^{(2)}_{x_2 \to z}), w_1(T^{(2)}_{z \to r})) \geq w_1(T^{(2)}_{x_2 \to r}) \tag{4.11}$$

We now build a tree $T^{''}$ from $T^{(2)}$ such that x_2 is stable : tree $T^{''}$ is a copy of $T^{(2)}$ except that the parent of x_2 is z. Vertex x_2 is stable in $T^{(2)}$.

Note that we build a sub-graph of G corresponding to a *dispute wheel* such that y_1 (resp. x_2) is u_j (resp. u_{j+1}) with $j \neq 0$. Now, it remains to find which vertex corresponds to vertex u_0.

By definition $T^{''}$ is not stable and x_1 and x_2 are stable. Tree $T^{''}$ can be transformed into tree $T^{(3)}$ so that all vertices in V_2 were stable. We transform this tree using the same way as we transform tree $T^{'}$ into $T^{(2)}$. So, for any vertex y in V_2 unstable in $T^{''}$, we have

$$w_2(T^{(3)}_{y \to r}) > w_2(T^{''}_{y \to r}) \tag{4.12}$$

Using the same arguments as previously, only x_1 is not stable in $T^{(3)}$. We will consider two cases:

Case 1.1: $w_2(T^{''}_{x_2 \to r}) < w_2(T^{(2)}_{x_2 \to r})$

Case 1.2: $w_2(T^{''}_{x_2 \to r}) > w_2(T^{(2)}_{x_2 \to r})$

Note that the case where $w_2(T^{''}_{x_2 \to r}) = w_2(T^{(2)}_{x_2 \to r})$ is impossible. If it is the case, $T^{''}$ would be stable. So there is a contradiction with the fact that all trees are not stable.

Case 1.1: Assume that $w_2(T''_{x_2 \to r}) < w_2(T^{(2)}_{x_2 \to r})$. By definition, all unstable vertex y in T'' are V_2 such that $x_2 \in T''_{y \to r}$ and hence $x_2 \in T^{(2)}_{y \to r}$. So we can deduce that for any unstable vertex y in $T^{(2)}$,

$$min(w_2(T^{(2)}_{y \to x_2}), w_2(T^{(2)}_{x_2 \to y_1}), w_2(T^{(2)}_{y_1 \to r})) > min(w_2(T''_{y \to x_2}), w_2(T''_{x_2 \to r}))$$

Hence, we get:

$$w_2(T^{(3)}_{y_1 \to r}) > min(w_2(T''_{y \to x_2}), w_2(T''_{x_2 \to r}))$$

Since x_1 is not stable in $T^{(3)}$, there is $y_2 \in (T''_{x_1 \to r})$ such that y_2 is not stable in T''. In other words, there is $z \in BR(x_1, T^{(3)})$ such that $min(w_1(T^{(3)}_{x_1 \to z}), w_1(T^{(3)}_{z \to r})) > min(w_1(T^{(3)}_{x_1 \to y_1}), w_1(T^{(3)}_{y_1 \to r}))$. By definition of stable node in $T^{(1)}$ in graph G_1, we have $w_1(T^{(1)}_{x_1 \to r}) = min(w_1(T^{(3)}_{x_1 \to z}), w_1(T^{(3)}_{z \to r}))$

In fact, G contains a *dispute wheel* where y_2 corresponds to u_0 of size 4. This fact contradicts with the fact that G does not contains a *dispute wheel*. We can conclude that this case is impossible.

Case 1.2: Assume that $w_2(T^{(2)}_{x_2 \to r}) < w_2(T''_{x_2 \to r})$.

Since x_2 is not stable in $T^{(2)}$, we have:

$$min(w_1(T^{('')}_{x_2 \to y_1}), w_1(T^{(2)}_{y_1 \to x_1}), w_1(T^{(2)}_{x_1 \to r})) > w_1(T^{(2)}_{x_2 \to r})$$

Since x_2 is stable in $T^{(1)}$, we get:

$$w_1(T^{(2)}_{x_2 \to r}) > min(w_1(T^{('')}_{x_2 \to y_1}), w_1(T^{(1)}_{y_1 \to r}))$$

In fact, G contains a *dispute wheel* where x_2 corresponds to u_0 of size 4. This fact contradicts with the fact G does not contains a *dispute wheel*. We can conclude that this case is impossible.

Case 2: These cases are also impossible. This fact can be shown by applying the same arguments as previously. So, all cases are impossible. So there is a contradiction with the fact that all spanning trees are not stable for metric w in G. □

Theorem 12. *If G contains no* dispute wheel*, there exists a stable maximum flow tree.*

Proof. The theorem is proved by induction on the cardinality α of V_1. The case when $\alpha = 2$ corresponds to Theorem 11 and for the inductive step can be shown using the same method in the proof of Theorem 11. □

4.7 Different Types of Equilibria in the System

Note that there is a difference between the existence of an equilibrium and the prospect of stabilization. In some settings, equilibrium may exist but the scheduler may choose actions to prevent the system from stabilizing, as we demonstrate in subsequent sections. Obviously only configurations that enable the existence of equilibria are relevant for executing algorithms whose purpose is to find them. We categorize the diffrent equilibria scenario below:

1. **No equilibrium:** Figure 4.1 is an example where there exists no equilibrium in the system.

2. **Single rooted equilibrium:** Figure 4.7 is an example where there exists exactly one rooted equilibrium in the system.

3. **Single non-rooted equilibrium:** Figure 4.5 is an example where there exists exactly one non-rooted equilibrium in the system, comprising of all the nodes except the root and the outer edges of the graph.

4. **More than one rooted equilibrium:** Multiple equilibria are also feasible in a system. Figure 4.3 shows an example where cleverly chosen weight configurations lead to multiple equilibria.

5. **Rooted and non-rooted equilibria:** There can be two equilibria in a system, one with the root and another without the root. Figure 4.4 shows an example. The four nodes except the root can constitute an equilibrium when their parent pointers do not point to the root but point to the next outer node and thus form a cycle, whereas if all the nodes except the root choose the root as their parent, then the system stabilizes to a rooted equilibrium.

4.8 Weak Stabilizing Algorithm for Maximum Flow Tree

All nodes in V have a *common goal* and a *private goal*. The common goal is as follows: starting from an arbitrary initial configuration, they collaborate with one another to form a *maximum flow tree* with a given node designated as the root. The private goal of each node is to maximize its *flow* value without violating the *flow* constraints *i.e.*, a node cannot push an amount of *flow* that is greater than its

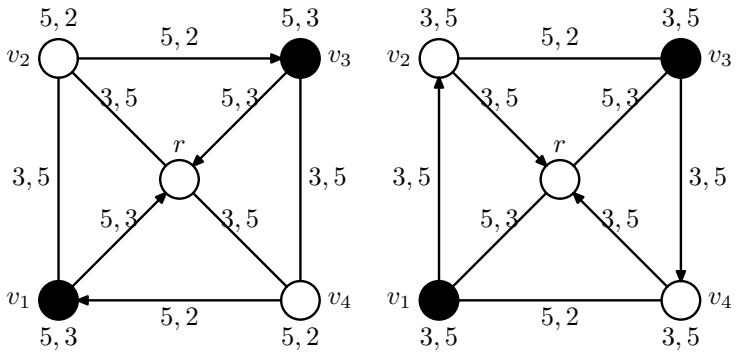

Figure 4.3: The two graphs are the same as their edge costs are the same. But it is to be noted that the equilibria in the two cases are different as the flow values of the nodes are different. So multiple equilibria are possible with the same setting.

parent's *flow* value or the *capacity* value of the edge that joins it to its parent.

We assume that each node i is aware of $N(i)$, the set of its neighbors (excluding i itself). Similarly, each node i is aware of the cost of each of its adjacent edges $e = (i, j) : j \in N(i)$. The cost of an edge e is a vector

$$w(e) = (w_1(e), w_2(e), w_3(e), \ldots, w_p(e))$$

where $w_k(e)$ denotes the cost of the edge e for a node of color k $(1 \geq k \geq p)$.

Also, i maintains two variables: $\pi(i)$ and $flow(i)$. The variable $\pi(i)$ denotes the parent of i in the maximum flow tree. By definition, for the root node r, $\pi(r)$ is non-existent. Every other node picks a neighboring node as its parent (*i.e.* the domain of the $\pi(i)$ variable is $N_G(i)$). The variable $flow(i)$ denotes the vector

$$flow(i) = (flow_1(i), flow_2(i), flow_3(i), \ldots, flow_p(i))$$

where $flow_k(i)$ denotes the flow for a node of the k^{th} color from node i to the root. By definition, $flow(r) =$ maximum capacity values for each color.

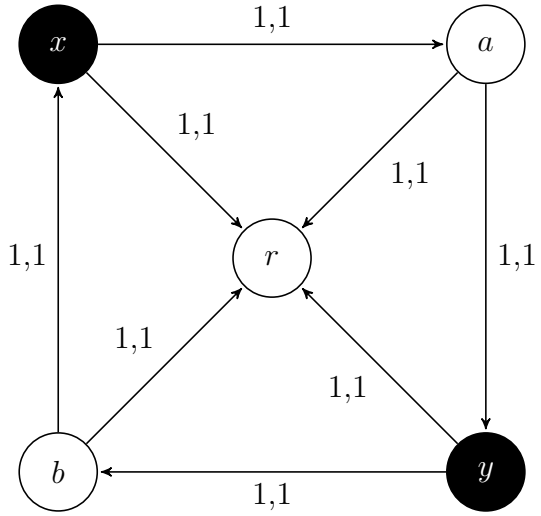

Figure 4.4: Two equilibria - one with the root, the other without the root

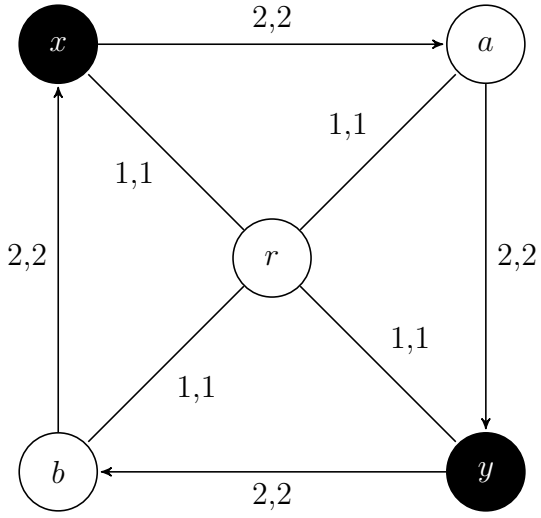

Figure 4.5: Single non-rooted equilibrium

The algorithm has three rules, executed by all the nodes except root r, when the guard of a node becomes true. These combined rules are able to find an equilibrium from configurations where there exists at least one equilibrium. Note that an equilibrium configuration may exist without the root node (See Figure 4.4).

Our flow algorithm is *weakly stabilizing* in the sense, in configurations described above for multiple equilibria, it may happen that the algorithm terminates without finding the rooted *maximum flow tree* equilibrium, which is the desired solution. Rather the algorithm may get stuck to the other equilibrium configuration which is the non-rooted equilibrium. But *weak stabilization* implies that there exists at least one computation that leads the system to the rooted *maximum flow tree* equilibrium configuration. So even though some action sequences will fail to find the desired solution, there is at least one action sequence which will be able to find the rooted *maximum flow tree* equilibrium.

The algorithm has three rules, for every node $i \neq r$. When the guard of node i of color k is true, it applies the following three rules in order:

R1 i of color k chooses a parent j from a set of nodes in its neighborhood such that i can push the maximum flow, provided i is not the parent of j. This parent-check condition is necessary to break short two-edge cycles where two neighboring nodes think each is other's parent.

R2 If i has only one neighbor j, then j becomes i's parent. This ensures the algorithm includes every node in the final *maximum flow tree*.

Algorithm 4.1 The Maximum Flow Tree Algorithm

R1 $fix\_parent(i_k) \equiv \pi(i_k) = j \in N_G(i_k) \wedge i_k \neq \pi(j) \wedge N_G(i_k) > 1 : \{\min[cap_k(i,j), flow_k(j)]\},\ k \in \mathcal{C}$

R2 $fix\_parent(i_k) \equiv \pi(i_k) = j \in N_G(i_k) \wedge i_k \neq \pi(j) \wedge N_G(i_k) = 1 : \{\min[cap_k(i,j), flow_k(j)]\},\ k \in \mathcal{C}$

R3 $fix\_flow_k(i) \equiv flow_k(i) = \min\left(flow_k(\pi(i), cap_k(i, \pi(i))\right),\ k \in \mathcal{C}$

R3 R3 means i adjusts its *flow* value after choosing its parent.

Fig 4.6 is the redrawing of Fig 4.1 to show how in case where there exists no equilibrium in the system; the algorithm, as expected, continues forever. The fact that one can always apply one of the last three rules at one or more nodes of $\mathcal{Z}$ after the first two rules makes the system oscillate forever.

The three set of numbers against each node denotes the flow values of that node for white and black colors respectively. It is to be noted that if we start with arbitrary initial values of the nodes (we chose $9, 9$ for convenience, but it could be any value), and let the scheduler choose any node whose guard is true, then the parent pointer for each node continually keeps changing. As a result, the flow values of each node also oscillates forever. In the above example, we choose the repetitive pattern of execution sequence $v_2, v_3, v_4, v_1, v_2, v_3, v_4, v_1$ ... to illustrate this. The x sign against an edge denotes at some point of the execution, it was (and again will be in the future) the parent and the last two flow values of each node denote the perpetual flow

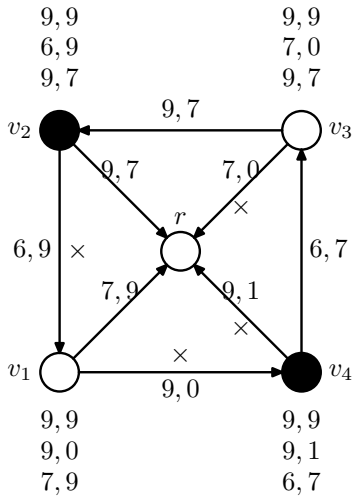

Figure 4.6: The graph $\mathcal{Z}$ with 5 nodes. Every node's initial flow value is (9,9). After rule 1) and 2) are applied, the repetitive pattern of execution sequence $v_2, v_3, v_4, v_1, v_2, v_3, v_4, v_1$ ... makes the system oscillate forever. There is no equilibrium for the above system no matter how the scheduler chooses the sequence of actions.

values of that particular node. For example, when v_2 chooses r as its parent, its flow value is given by $9, 7$ but when it chooses v_1 as its parent, its flow value is $6, 9$. This is true for the other nodes too except the root.

Fig 4.7 shows an instance of equilibrium applying the algorithm in a different system where equilibrium exists.

4.9 Proof of Correctness

Theorem 13 (Partial Correctness)**.** *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements. Let w be a function on $E \rightarrow N^p$. If no node has enabled rules and there is a single rooted equilibrium, the structure induced by the parent variables of each node induces a stable tree T rooted at r for metric w.*

Proof. It is easy to show that the induced structure is a tree. Due to the fact that

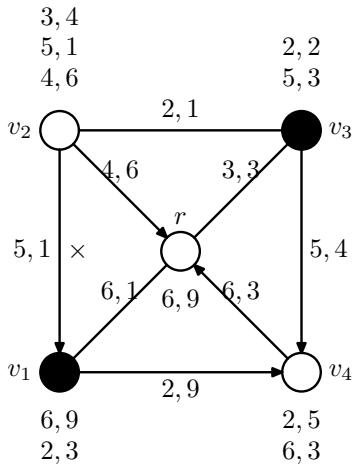

Figure 4.7: Example of the algorithm finding the equilibrium in a system. In the above graph, the execution sequence is v_3, v_4, v_2, v_1, v_2. After five moves, stabilization is achieved. We chose some arbitrary initial flow values for the nodes to begin with (which are $(6, 9); (3, 4); (2, 2); (2, 5)$ respectively for v_1, v_2, v_3, v_4). The x against an edge between v_2 and v_1 indicates that v_2 initially chose v_1 as its parent but later changed its parent. The updated flow values of the nodes are written against each node.

the parent variables always point to a neighbor (except r), the structure consists of either a tree or a connected component consisting of a tree and a set of circuits. Assume for the purpose of contradiction that there exists at least one such circuit. As w only has costs in N, all flow values of nodes are strictly positive. As no rule is enabled, this implies that all flow values are OK, and thus each flow value is either equal or less than the flow value of the parent. Due to the well founded property of the"lower than" relation on (positive) integers, such a circuit is impossible. Hence, by contradiction we infer that the induced structure is a tree.

Now, we prove that the obtained tree T is stable for metric w. As no rules are enabled, this means that the flow variables are all properly computed from the local metric information. By induction on the flow path to the root r, this means that the

flow variables contain the correct values for the cost of every path towards the root according to metric w. Now, the parent changing rules are also not enabled. This implies that the parent variables are actually selecting the best neighbor according to the color k of node $i \in V_k$. This best neighbor being selected according to the metric w, we conclude that the tree T that is induced by the parent variables is a stable tree for metric w. □

Theorem 14. *Let $G = (V, E)$ be a graph and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements. Let w be a function on $E \to N^p$. If there exists a configuration where the parent variables induce at least a single maximum flow tree to the root, then there exists an execution of the flow algorithm that leads to the* maximum flow tree *configuration with a central demon.*

Proof. Let us consider the execution of different rules of the algorithm. R3 does flow adjustment, so nothing in terms of parent selection alters for applying this rule.

Now let us assume all nodes that have their fix_flow (R3) rule enabled execute it, and that nodes that have their parent fixing rule don't execute it. Since the π variables induce a tree to the root, only a fixed number of fix_flow can be executed. If in this configuration, no node has a rule to execute, the configuration is terminal. Otherwise, this means that there exists at least one node that is willing to change its parent. When one node changes its parent, there exists an execution in which all nodes in its subtree update their flow value and no other parent fixing rule is executed. By the acyclic nature of the tree, there is a finite number of such moves.

The only configuration where repetition can persist in the system is where there

are equilibria in the system without a tree configuration. Then the acyclic property will not hold anymore and R1 can lead the system to a non-rooted equilibrium which involves dispute wheels. Otherwise, following the argument above, we can conclude that R1 always finds the rooted maximum flow tree if there exists at least one such tree in the system.

It is to be noted that R2 does not anyway conflict with finding the rooted tree equilibrium configuration. If there is a node whose only neighbor is the root, then the root is selected as the parent. Otherwise the node in question is a leaf node and applying R2 just ensures that all nodes are included in the final *maximum flow tree*.

Hence there exists a configuration where the π variables induce at least a single maximum flow tree to the root implies that there exists an execution of the flow algorithm that that leads to the *maximum flow tree* configuration with a central demon. □

Theorem 15 (Weak Stabilization)**.** *Let $G = (V, E)$ be a graph containing no dispute wheel and r a node of V. Let $\mathcal{P}$ be a partition of V with p elements. Let w be a function on $E \rightarrow N^p$. The flow algorithm is weakly self-stabilizing for the stable tree construction problem according to metric w.*

Combining the results from Theorem 13 and Theorem 14, we get the result of the above theorem. Weak stabilization is in contrast with traditional stabilization (call it *strong stabilization*) that allows the demon to pick an arbitrary process with an enabled guard, and schedules its action. Strong stabilization requires that *all computations* starting from an arbitrary configuration lead the system to an equilib-

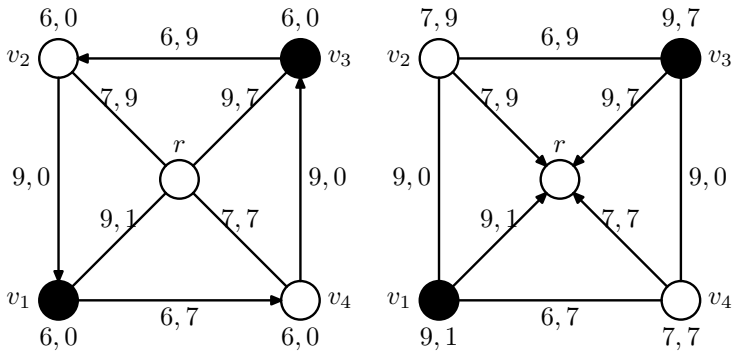

Figure 4.8: An example to show that a synchronous demon can play the role of an adversary and the configurations can repeat forever. If all the four nodes simultaneously make moves, then the two configurations alternately repeat. It is to be noted there is an equilibrium in the above system though, which is reachable using a central demon from the first configuration, with the execution sequence v_3, v_4, v_1, v_2.

rium configuration. Of course, our (weakly) self-stabilizing solution does not exclude the existence of (strongly) self-stabilizing solutions, but proving the existence of such solution is an open question.

Observation 7. *Stabilization may not be feasible if at the same time, more than one process make moves i.e., a synchronous demon can play the role of an adversary and the configurations can repeat infinitely (see Fig 4.8).*

4.10 Discussions

The paradigm of selfish stabilization can easily be extended in several ways. First, it can easily be extended to systems involving more than two competing groups, in the extreme case, each process caring for itself and no one else. Second, the metrics used here - simple additive metric and strictly monotonic metric could be replaced by some suitable other metric.

CHAPTER 5
PROBABILISTIC FAULT-CONTAINMENT

5.1 Introduction

Research on fine tuning stabilization properties has received attention for more than a decade. This chapter addresses the problem of fault-containment, and presents a *probabilistic* algorithm for the persistent-bit protocol. The algorithm confines the effect of any single fault to the immediate neighborhood of the faulty process, with an expected recovery time of $O(\Delta^3)$, where Δ is the maximum degree in the network - while guaranteeing eventual recovery from arbitrary initial configurations.

We follow the general computational model for fault-containment described in chapter 2. A basic problem for state-corrupting faults is the *persistent-bit* problem: how to maintain the value of a single bit across a network of processes, when state corruption occurs. Formally, the problem is defined as follows: each node i maintains an externally observable output bit $v(i)$. In a legal configuration $LC \equiv \forall i, j : v(i) = v(j)$. Failures can corrupt the value of v, but a stabilizing system must guarantee that a legal configuration is eventually restored.

5.2 Contributions

Our solution to the persistent-bit protocol is weakly fault containing with a randomized scheduler. *Weak fault-containment* means that from all single failures, the expected recovery time of the transformed version is dependent only on the degree of the nodes, and independent of the network size. Furthermore, observable changes

are confined to only the immediate neighbors of the faulty processes with a high probability. In addition to the weak fault-containment property, the system recovers from all k-faulty ($k > 1$) configurations in an expected number of $O(k.n)$ steps for an array of n processes.

Why should anyone care about an algorithm that allows the neighbors to be contaminated, when better solutions are available [25]? The answer lies in the small *fault-gap*. Once a fault-containing system recovers from a single failure in constant time, the *fault-gap* means how much time will elapse before the system becomes ready to recover from the next single failure with the same efficiency. The dramatic growth of the network size raises the probability of failures, and many solutions to fault-containment that we know of achieve a fault-gap of $O(n)$ or worse. As a result, when two single faults occur relatively quickly, the system fails to provide the guarantee of efficient recovery, and in fact the second single failure may require $O(n)$ (or higher) time for recovery depending on the problem to be solved. This seriously undermines the availability of the fault-free system. Our solution guarantees that the fault-gap depends only on the degree of the nodes, and is independent of the size of the network. This will significantly increase the *availability* of the fault-free system. In fact, our solution handles the recovery from simultaneous failures at nodes that are distance-3 or more apart with the same efficiency, as long as such failures occur at intervals of $O(\Delta^3)$ or higher.

5.3 Persistent-bit Protocol

For the sake of exposition, we consider the case of *persistent-bit* protocol, in which a set of processes maintains the value of a replicated bit $v \in \{0, 1\}$ across a connected network. There are two distinct legal configurations: all 0's or all 1's. The following protocol is weakly stabilizing:

Algorithm 5.1 Persistent-bit protocol: Program for process i

A1 **do** $\exists j \in N(i) : v(j) \neq v(i) \rightarrow v(i) := v(j)$ **od**

Note. With a deterministic demon, it is easy to verify that the protocol is not stabilizing. With a *weakly fair scheduler* and a *randomized demon*, the protocol stabilizes to a legal configuration with probability 1, but it is not fault containing. Therefore in the next section, we present a probabilistic approach to make the protocol fault containing.

Lemma 4. *The persistent-bit protocol is not fault-containing with a randomized demon.*

Proof. Consider a linear array of processes numbered $0, 1, \ldots, n-1$ from left to right, and assume that initially $\forall i : v(i) = 1$ holds. Let a failure of process 0 change $v(0)$ to 0. With this as the starting state, the computation can be reduced to a run of *gambler's ruin*[1]: whenever a process with $v = 0$ executes an action, the boundary between the

[1]The original study is by Coolidge[32] in 1909, where he showed that if two gamblers

dissimilar values of v shifts to the left, and whenever a process with $v = 1$ executes an action, the boundary moves to the right. The game is over when the system reaches LC, and per [32] the expected number of moves needed is $(1 \times n - 1)$, *i*.e. $O(n)$. Thus, the protocol is *not* fault-containing. □

5.4 Probabilistic Algorithm for Fault-Containment

To make the protocol fault-containing, we add to each process i a secondary variable $x(i)$ whose domain is the set of non-negative integers. In a way, $x(i)$ will reflect the priority of process i in executing an action to update $v(i)$. Process i will update $v(i)$, when the following three conditions hold:

1. The randomized scheduler chooses i,
2. $\exists j \in N(i) : v(j) \neq v(i)$, and
3. $\forall j \in N(i) : x(i) \geq x(j)$.

After updating $v(i)$, process i will increase $x(i)$ to $max\ \{x(j) : j \in N_i\} + m$, where $m > 0$. In case only the first two conditions hold, but not the third, process i will increment the value of $x(i)$ by 1, and leave $v(i)$ unchanged. Algorithm 5.2 shows the modified protocol:

Observe that once a process i updates $v(i)$, it becomes difficult for its neighbors to change their v-values, since their x-values will lag behind that of i. The larger is

---

start with capitals of x and $N - x$, and each fair coin toss transfers a dollar from one to the other depending on the outcome of the toss, then the expected number of steps to finish the game is $x.(N - x)$.

Algorithm 5.2 Probabilistic fault-containing algorithm: Program for process i

A1 $\exists j \in N(i) : v(j) \neq v(i) \wedge \forall k \in N(i) : x(i) \geq x(k) \rightarrow v(i) := v(j); x(i) := max\{x(k) : k \in N(i)\} + m$

A2 $\exists j \in N(i) : v(j) \neq v(i) \wedge \exists\, k \in N(i) : x(i) < x(k) \rightarrow x(i) := x(i) + 1$

A3 $\forall j \in N(i) : v(j) \neq v(i) \rightarrow v(i) := v(j)$

the value of m, the greater is the difficulty. A neighbor j of i will be able to update $v(j)$ only if it is chosen by the random scheduler m times, without choosing i even once. On the other hand, it becomes easier for i to update $v(i)$ again in the near future.

Failures can not only corrupt v, but also corrupt x. Assume that $LC \equiv \forall j : v(j) = 1$, a single failure at process i changes $v(i)$ to 0, and $x(i)$ to some unknown value. If $\forall j \in N(i) : x(i) > x(j)$ then process i is likely to change its $v(i)$ soon again. As a result, the fault is contained in a small number of steps, and the contamination number is one. However, a smart adversary injecting the failure at process i is likely to set $x(i)$ to the smallest value (*i*.e. 0). This makes the neighbors of process i better candidates for changing their v, before process i executes a move to complete the recovery. However, it also raises the x-values of these neighbors of i above those of *their* neighbors. In order that the fault percolates to a node at distance-2 from the faulty process i, such a distance-2 node has to be chosen by the scheduler at least m times, without choosing its neighboring distance-1 node even once. With a large

value of m, the probability of such an event is very low. This explains the mechanism of containment. In the mean time, the condition $\forall j \in N(i) : v(j) = 1$ is likely to hold several times. If on one such occasion the faulty process is chosen by the random scheduler (the third action, note that its guard does not depend on x) then $v(i)$ will change to 1, and the recovery will be complete.

5.5 Results

We begin with an analysis of the spatial containment. Assume that all nodes have a degree Δ. Then the following theorem holds:

Theorem 16. *As $m \to \infty$, the effect of a single failure is restricted to only the immediate neighbors of the faulty process.*

Proof. Suppose the faulty process has n_1 neighbors and the contaminated process has n_2 neighbors that are distance-2 neighbors of the faulty process. The probability that a distance-2 neighbor is contaminated is largest, when only one distance-1 process is contaminated, then only one neighbor of that contaminated distance-1 process (which is a distance-2 neighbor of the faulty process) is contaminated. The probability of one distance-1 neighbor being contaminated is $\frac{n_1}{n_1+1}$. To contaminate a distance-2 neighbor, the scheduler must select the specific process m times. So the probability of one distance-2 neighbor being contaminated is $\frac{1}{(n_1+n_2+1)^m}$. Therefore, after a node becomes faulty, the probability that some distance-2 neighbor of the faulty process becomes contaminated is $\frac{n_1}{n_1+1} \times n_2 \times \frac{1}{(n_1+n_2+1)^m}$. By choosing a large value of m, this probability can be made as small as possible. □

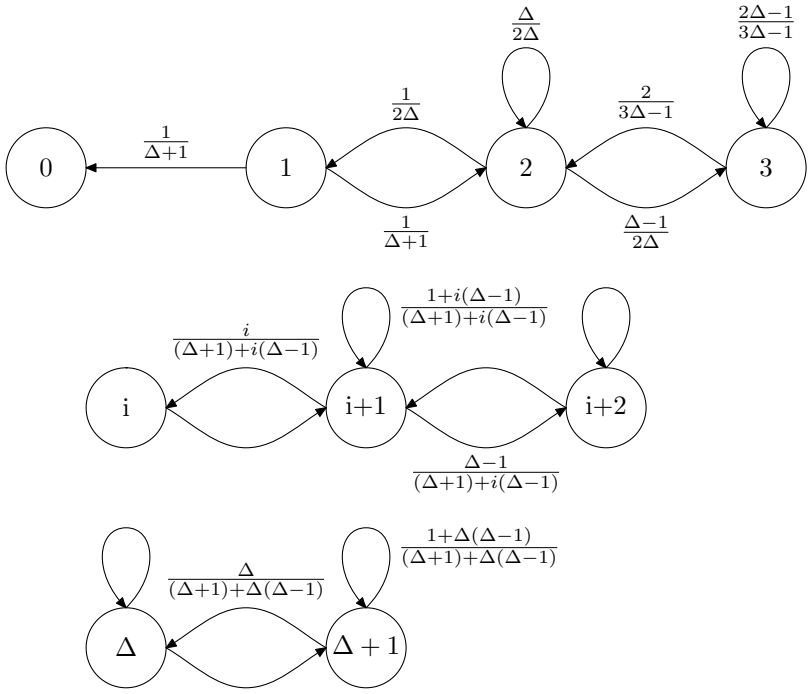

Figure 5.1: Each node is a state corresponding to the size of the faulty region, and the label on each edge represents the probability of the corresponding state transition.

Theorem 17. *If $\Delta << m$ then the expected number of steps needed to contain a single fault is $O(\Delta^3)$.*

Proof. Assume m (or M for the bounded solution presented later) is very large, so the error is unlikely to propagate to the distance-2 neighbors of the faulty process. In such a scenario, it is sufficient to consider the case when the error propagates to the immediate neighbors of the 1-faulty process. The state transfer diagram showing the transitions among various faulty configurations is shown in Figure 5.1.

As the error will propagate at most to the immediate neighbors, the system can have at most Δ+1 errors. We use $p_{i,j}$ to denote the probability that the number

of the errors changes from i to j. So the probabilities are listed below:

$$p_{1,0} = \frac{1}{\Delta + 1} \tag{5.1}$$

$$p_{2,1} = \frac{1}{2\Delta} \tag{5.2}$$

$$p_{2,2} = \frac{\Delta}{2\Delta} \tag{5.3}$$

$$p_{1,2} = \frac{1}{\Delta + 1} \tag{5.4}$$

and for $3 \leq i \leq \Delta - 1$

$$p_{i,i+1} = \frac{i}{(\Delta + 1) + i(\Delta - 1)} \tag{5.5}$$

$$p_{i+1,i+1} = \frac{1 + i(\Delta - 1)}{(\Delta + 1) + i(\Delta - 1)} \tag{5.6}$$

$$p_{i+1,i+2} = \frac{\Delta - 1}{(\Delta + 1) + i(\Delta - 1)} \tag{5.7}$$

and

$$p_{\Delta+1,\Delta} = \frac{\Delta - 1}{(\Delta + 1) + \Delta(\Delta - 1)} \tag{5.8}$$

$$p_{\Delta+1,\Delta+1} = \frac{1 + \Delta(\Delta - 1)}{(\Delta + 1) + \Delta(\Delta - 1)} \tag{5.9}$$

We use $P[X]$ to denote the probability that the system recovers using x moves. The expected number of moves needed is

$$\begin{aligned} E &= 1 \times P[1] + 2 \times P[2] + 3 \times P[3] + \cdots \\ &= \sum_{x=1}^{\infty} XP[X]. \end{aligned}$$

We can calculate $P[X]$ as following using (5.1), (5.4), (5.2):

$$P[1] = p_{1,0} = \frac{1}{\Delta + 1} \tag{5.10}$$

$$P[2] = 0 \tag{5.11}$$

$$P[3] = p_{1,2}p_{2,1}p_{1,0} = \frac{1}{\Delta+1}\frac{1}{2\Delta}\frac{1}{\Delta+1} = \frac{1}{2\Delta(\Delta+1)^2} \tag{5.12}$$

$$P[4] = p_{1,2}p_{2,2}p_{2,1}p_{1,0} = \frac{1}{\Delta+1}\frac{\Delta}{2\Delta}\frac{1}{2\Delta}\frac{1}{\Delta+1} \tag{5.13}$$

and for $n \geq 1$, we get the following recursive function of $P[2n+2]$ using $P[2n+1]$:

$$P[2n+2] = 2\sum_{j=2}^{m} p_{j,j}P[2n+1] \tag{5.14}$$

If $n+1 < \Delta+1$ then m=n. If $n+1 > \Delta+1$ then $m = \Delta+1$. This is because if $n+1 < \Delta+1$, using $2n+2$ steps, the system can reach at most the $n+1$-th state. So the repeated moves can happen in any state within the n-th state. But if $n+1 > \Delta+1$, the system can move through all the states, so the repeated moves can happen anywhere within the $\Delta+1$ states.

We can also write $P[2n+3]$ using $P[2n+1]$ as:

$$P[2n+3] = 2\sum_{j=2}^{m}\sum_{i=2}^{m} p_{j,j}p_{i,i}P[2n+1] + 2\sum_{k=2}^{m'} p_{i,i+1}p_{i+1,i}P[2n+1] \tag{5.15}$$

The values of m and m' are the same as in (5.14). The system can have either two additional moves that do not change the current state, or the system can first reach a state in one step and come back in the next step. So we substitute $P[1], P[2], P[3], P[4]$ using (5.10), (5.11), (5.12), (5.13), (5.14), (5.15) in (5.10) and let $p_{max} =$

$max\{p_{i,j}, \forall i, \forall j\}$:

$$
\begin{aligned}
E &= 1 \times P[1] + 2 \times P[2] + 3 \times P[3] + 4 \times P[4] \\
&+ \sum_{n=2}^{\infty}\{(2n+1)P[2n+1] + (2n+2)P[2n+2] + (2n+3)P[2n+3]\} \\
&= 1 \times \frac{1}{\Delta+1} + 2 \times 0 + 3 \times \frac{1}{2\Delta(\Delta+1)^2} + 4 \times \frac{1}{4\Delta(\Delta+1)^2} \\
&+ \sum_{n=2}^{\infty}\{(2n+1) + 2(2n+2)\sum_{j=2}^{m}\sum_{i=2}^{m} p_{i,i}p_{j,j} + (2n+3)(2\sum_{i=2}^{m'} p_{i,i+1}p_{i+1,i} \\
&+ 2\sum_{j=2}^{m''}\sum_{i=2}^{m''} p_{i,i}p_{j,j})\} \times P[2n+1] \\
&< O(\frac{1}{\Delta^2}) + \sum_{n=2}^{\infty}\{(2n+1)p_{max}^{2n+1} + 2(2n+2)\Delta^2 p_{max}^{2n+2} + (2n+3)(2\Delta+2\Delta^2)p_{max}^{2n+3}\} \\
&= \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1} + 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2} + (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3} + O(\frac{1}{\Delta^2}).
\end{aligned}
$$

Let $T_1 = \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1}, T_2 = 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2}, T_3 = (2\Delta + 2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3}$. As T_3's order is the same as T_2 and larger than T_1, we just need to calculate T_3.

$$
\begin{aligned}
T_3 &= (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3} \\
&= (2\Delta+2\Delta^2)(2p_{max}^3\sum_{n=2}^{\infty} np_{max}^{2n} + 3p_{max}\sum_{n=2}^{\infty} p_{max}^{2n}) \\
&= (2\Delta+2\Delta^2)[2p_{max}^3(\frac{2p_{max}^4}{1-p_{max}^2} + p_{max}^6) + 3p_{max}^3\frac{1}{1-p_{max}^2}] \\
&= (2\Delta+2\Delta^2)\frac{4p_{max}^7 + 2p_{max}^3p_{max}^6(1-p_{max}^2) + 3p_{max}^3}{1-p_{max}^2}.
\end{aligned}
$$

Let $p_{max} = \frac{a}{\Delta+a}, a = 1 + \Delta(\Delta - 1)$:

$$\begin{aligned} T_3 &= (2\Delta + 2\Delta^2)\frac{4a^7(\Delta+a)^2 + 2a^9 + 3a^3(\Delta+a)^6}{(\Delta^2 + 2\Delta a)(\Delta + a^7)} \\ &= O(\Delta^2) \times \frac{O(a^9)}{O(\Delta^3)O(a^7)} \\ &= \frac{O(a^2)}{O(\Delta)} \\ &= O(\Delta^3). \end{aligned}$$

So we get

$$E = O(\frac{1}{\Delta^2}) + O(\Delta^3) = O(\Delta^3) \tag{5.16}$$

□

When the graph is dense, *i.e.* $\Delta = O(n)$, the containment time is not independent of the size of the network anymore. However, spatial containment property still holds. The more dense the graph is, the smaller is the contamination number. As $m \to \infty$, contamination number tends to 1. Below, we separately analyze the extreme case of a dense topology: a completely connected graph.

Theorem 18. *For a completely connected graph, then the contamination number is* 1 *as* $m \to \infty$.

Proof. At least one neighbor j of the faulty process i is likely to update $v(j)$, and raise $x(j)$ at least m steps above the x-values of the rest. To prevent a second neighbor k from updating $v(k)$, the system must recover to L before the scheduler chooses the neighbor k at least m times, without choosing j even once. We use $P\{n_0, n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_{\Delta-1}\}$ to denote the probability that $\forall i, 1 \leq i \leq \Delta$,

node i is chosen n_i times. So the probability of node k being chosen m times before j being chosen even once is:

$$P\{n_0, n_1, n_2, \ldots, n_{j-1}, n_j = 0, n_{j+1}, \ldots, n_k = m, \ldots, n_{\Delta-1}\} \tag{5.17}$$
$$\forall i, 0 \leq i \leq \Delta - 1 \wedge i \neq j, n_i \leq m.$$

With increasing $n_i, 1 \leq i \leq \Delta \wedge i \neq j$ (5.17) is decreasing (see (Lemma 5 for the proof). So the above probability is maximum when node k is consecutively chosen m times, and other nodes are never chosen. The maximum probability is:

$$P\{0, 0, \ldots, 0, m, 0, \ldots, 0, 0\} = \frac{1}{\Delta^m} \tag{5.18}$$

Since $n_i, 0 \leq i \leq \Delta - 1 \wedge i \neq j$ can be any value between 0 and m, and there are totally $m^{\Delta-2}$ possible situations, we apply the maximum estimate to each such case. As a result, the probability of the system having two contaminated processes is no larger than $\frac{m^{\Delta-2}}{\Delta^m}$, which approaches 0 as m approaches ∞. □

Theorem 19. *For a completely connected graph, the containment time is $O(n^3)$, where n is the number of nodes in the graph.*

Proof. Following the same notation as in the previous theorem, We calculate the probabilities on a complete graph.

The expectation formula is then given by the following:

$$\begin{aligned} E(X) &= \sum_{X=1}^{\infty} X \times P(X) \\ &= 1 \times P(1) + 2 \times P(2) + \cdots + (i+1) \times P(i+1) + \cdots \end{aligned}$$

Now we know:

$$\begin{aligned}
P(1) &= \frac{1}{n} \\
P(2) &= 0 \\
P(3) &= \frac{1}{n} \times \frac{n-1}{n} \times \frac{1}{n} \\
P(4) &= \frac{n-1}{n} \times \frac{n-1}{n} \times \frac{1}{n} \times \frac{1}{n} \\
P(5) &= (\frac{n-1}{n} \times \frac{1}{n})^2 \times \frac{1}{n} + \frac{n-1}{n} \times (\frac{n-1}{n})^2 \times \frac{1}{n} \times \frac{1}{n}
\end{aligned}$$

We calculate the two general terms $P(i+2)$ and $P(i+3)$ in terms of $P(i+1)$:

$$\begin{aligned}
P(i+2) &= (i-2) \times \frac{n-1}{n} \times P(i+1) \\
P(i+3) &= [C^2_{i+1} + C^1_{i+1}] \times (\frac{n-1}{n})^2 \times P(i+1)
\end{aligned}$$

Note that the combinatorial terms appear because of the fact, when we try to insert two steps, it may occur in two ways: $\frac{n-1}{n} \times \frac{1}{n}$ accounts for one case, and $(\frac{n-1}{n})^2$ accounts for the other. As $(\frac{n-1}{n})^2$ is the worst case scenario, we need to consider only that.

From term $P(6)$ onwards, we can proceed using the general terms $P(i+1)$, $P(i+2)$, $P(i+2)$:

$$\begin{aligned}
&\sum_{i=5}^{\infty}[P(i+1) + P(i+2) + P(i+3)] \\
&\qquad = \sum_{i=5}^{\infty} P(i+1)\{1 + (i-2) \times \tfrac{n-1}{n} + [C^2_{i+1} + C^1_{i+1}] \times (\tfrac{n-1}{n})^2\}
\end{aligned}$$

Using the inequality

$$P(i+1) \leq (\frac{n-1}{n})^{i+1} \tag{5.19}$$

We get:

$$\sum_{i=5}^{\infty}[P(i+1)+P(i+2)+P(i+3)]$$

$$\leq \sum_{i=5}^{\infty}\{1+(i-2)\times\frac{n-1}{n}+[C_{i+1}^{2}+C_{i+1}^{1}]\times(\frac{n-1}{n})^{2}\}\times(\frac{n-1}{n})^{i+1}$$

Let $q=\frac{n-1}{n}$ and $m=i-2$ and

$$z=\sum_{i=5}^{\infty}\{1+(i-2)\times\frac{n-1}{n}+[C_{i+1}^{2}+C_{i+1}^{1}]\times(\frac{n-1}{n})^{2}\}\times(\frac{n-1}{n})^{i+1}$$

So,

$$z=\sum_{m=3}^{\infty}q^{m+3}+\sum_{m=3}^{\infty}m\times q^{m+4}+\sum_{m=3}^{\infty}\frac{(m+3)\times(m+2)}{2}\times q^{m+5}+\sum_{m=3}^{\infty}(m+3)\times q^{m+5} \tag{5.20}$$

Simplifying, we get the first term is $O(n)$, the second term is $O(n^2)$, and the third and fourth terms combined are $O(n^3)$. So, for a completely connected graph, the containment time is $O(n^3)$, where n is the number of nodes in the graph. □

Lemma 5. *The probability that node k is chosen m times before node j is chosen once is maximum when node k is consecutively chosen m times and other nodes are never chosen.*

Proof. The probability that node k is chosen m times before node j is chosen once is:

$$\begin{aligned} P\{n_k = m, n_j = 0\} &= \sum_{i=1 \wedge i \neq j \wedge i \neq k}^{\Delta} \sum_{n_i=0}^{m} P\{n_1, n_2, \ldots, n_j = 0, \ldots, n_k = m, \ldots, n_\Delta\} \\ &= \begin{pmatrix} \sum_{i=1}^{\Delta} n_i \\ n_1 \end{pmatrix} \begin{pmatrix} \sum_{i=2}^{\Delta} n_i \\ n_2 \end{pmatrix} \cdots \begin{pmatrix} \sum_{i=j}^{\Delta} n_i \\ 0 \end{pmatrix} \cdots \\ &\times \begin{pmatrix} \sum_{i=k}^{\Delta} n_i \\ m \end{pmatrix} \cdots \begin{pmatrix} \sum_{i=\Delta}^{\Delta} n_i \\ n_\Delta \end{pmatrix} \left(\frac{1}{\Delta}\right)^{\sum_{i=1}^{\Delta} n_i} \\ &= l \times \left(\frac{1}{\Delta}\right)^q. \end{aligned}$$

If we increase any $n_i, 1 \leq i \leq \Delta \wedge i \neq j \wedge i \neq k$ to $n_i + 1$, we can see this selection process as follows: first do the selection in the same way as before we increase n_i, then choose any one of the eligible process. There are in all $\Delta - 1$ processes that can be chosen for the last step, and the probability of choosing any one of them is $\frac{1}{\Delta}$, so the probability will be $l \times (\Delta - 1) \times \left(\frac{1}{\Delta}\right)^{q+1}$ and this is smaller than $l \times \left(\frac{1}{\Delta}\right)^q$. So the probability that node k is chosen m times before choosing node j chosen once will decrease as n_i increases. □

The mechanism will reveal that a high clustering coefficient limits the probability of contamination to only a small fraction of the distance-1 neighbors. The completely connected graph exhibits an extreme form of this property.

5.5.1 Computing the Availability

An interesting aspect of the proposed algorithm is that $LC_s = true$, thus there is no overhead for stabilizing the secondary variables. So, LC holds as soon as LC_p

holds. This leads to the following theorem:

Theorem 20. *For single failures, the fault-gap equals the containment time.*

As a consequence of this, within an expected time of $O(\Delta^3)$ after each single failure, the system is ready to withstand the next single failure with the same efficiency. Furthermore, since only the distance-1 neighbors are contaminated with high probability, the proposed algorithm enables the system to recover from all concurrent failures of nodes that are distance-3 or more apart with the same efficiency. This significantly increases the availability of the system compared to existing solutions that we know of.

5.6 A Bounded Solution

A drawback of the proposed solution is that the x-variables grow in an unbounded manner, and it affects the implementability of the protocol. To address this, we will now transform the solution into one that relies on bounded variables only.

In Algorithm 5.2, when a process i executes action 1, it raises the value of $x(i)$ so that $\forall j \in N(i) : x(j) < x(i)$ holds. This makes process i a local leader, and when the local leader is chosen by the scheduler, it immediately executes its action to update $v(i)$. Let us set an upper bound $M-1$ for x, where M is an odd integer and $M > 1$ Furthermore, we let actions 1 and 3 increment $x \ mod \ M$. To make $x(j)$ "less than" $x(i)$, we have to define the *l*ess than operation $\prec$ appropriately. We define it as follows:

if $x(j) \in \{x(i)+1 \ mod \ M, \ x(i)+2 \ mod \ M, \ \ldots, \ x(i)+\frac{M-1}{2} \ mod \ M\}$

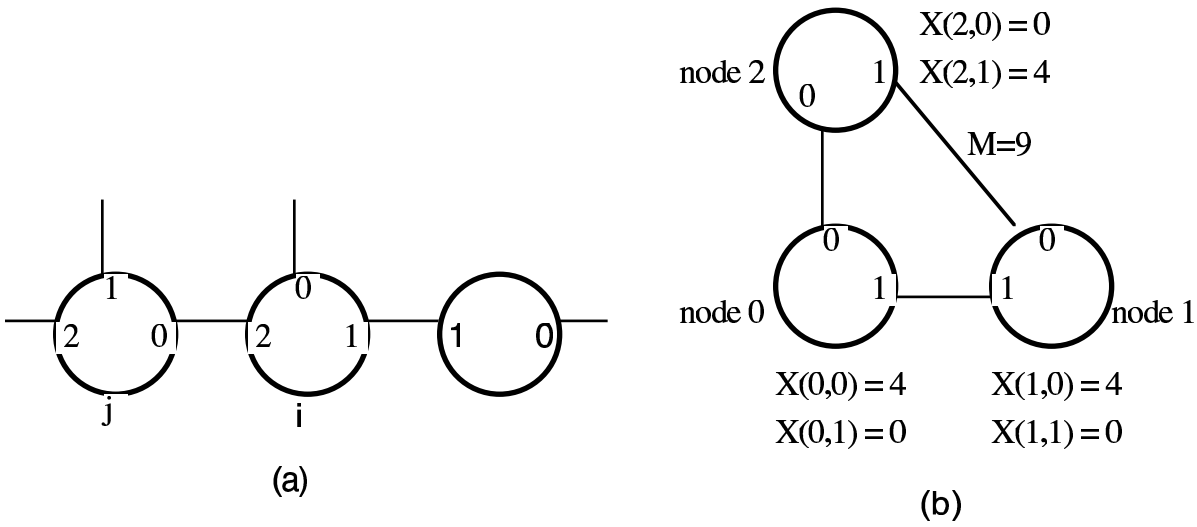

Figure 5.2: The bounded solution: (a) Identifying the ports of a node, (b) With $M = 9$, there is no local leader here.

then $x(i) \prec x(j)$ **else** $x(j) \prec x(i)$

Clearly, $\prec$ is not transitive. In order that the condition $\forall j \in N(i) : x(j) \prec x(i)$ holds, we will treat each x as a vector (and denote it henceforth by X) with Δ elements $0, 1, \ldots, \Delta - 1$. Let the k^{th} port of process i be connected to the l^{th} port of process j (Fig. 2(a)). We denote this by $((i,k),(j,l)) \in E$. A process i will execute action 1 when

$\exists j \in N(i) : v(j) \neq v(i), \wedge$

$\forall k \in N(i) : ((i,u),(k,w)) \in E,\ X(k,w) \prec X(i,u).$

We also modify the last part of action 1 as:

$\forall k \in N(i) : ((i,u),(j,w)) \in E,\ X(i,u) := X(j,w) + \frac{M-1}{2}\ mod\ M$

The above modification explicitly forces the condition $X(k,w) \prec X(i,u)$ across each

edge (i, k) of node i connecting to a neighbor, and establishes process i as a local leader, by setting the components of X at a maximum distance "above" those of its neighbors. Algorithm 5.3 shows the bounded version of Algorithm 5.2. The purpose of the second action is to let the non-leaders gradually "catch up" with a neighboring local leader, and is an adaptation of action 2 in Algorithm 5.2.

Algorithm 5.3 The bounded solution: Program for process i

A1 $\exists j \in N(i) : v(j) \neq v(i) \wedge \forall k \in N(i) : ((i,u),(k,w)) \in E,\ X(k,w) \prec X(i,u) \rightarrow$
$v(i) := v(j);\ \forall k \in N(i) : ((i,u),(j,w)) \in E,\ X(i,u) := X(j,w) + \frac{M-1}{2}\ mod\ M$

A2 $\exists j \in N(i) : v(j) \neq v(i) \wedge \exists k \in N(i) : ((i,u),(k,w)) \in E \wedge X(i,u) \prec X(k,w)$
$\rightarrow X(i,u) := X(i,u) + 1\ mod\ M$

A3 $\forall j \in N(i) : v(j) \neq v(i) \rightarrow v(i) := v(j)$

Using the modified interpretation of the "less than" relation $\prec$, and by replacing m by $\frac{M-1}{2}$, Algorithm 5.3 becomes semantically equivalent to Algorithm 5.2. However, by converting x into a vector X, there is no guarantee that there will be always be a local leader (Figure 5.2) ready to execute action 1. If the initial configuration is 1-faulty, and the scheduler chooses the faulty process, then this is not a concern, since action 3 does not rely on the x values at all. However, this may be an issue when the system starts from a k-faulty configuration and $k > 1$ We close our arguments by discussing the impossibility of deadlock in Algorithm 5.3.

Theorem 21. *Algorithm 5.3 guarantees that starting from any initial configuration, eventually some node is elected as a local leader.*

Proof. Assume that the system starts with a configuration where there is no local leader eligible to execute action 1. The possibility of some non-leader becoming a local leader requires that the scheduler chooses it $\frac{M-1}{2}$ times without choosing a neighbor even once. The probability of this event is $2^{-\frac{M-1}{2}}$. □

Once a local leader is elected, there always exists at least one local leader until the recovery is complete. Thus the time $2^{\frac{M-1}{2}}$ is an additional start-up cost that needs to be added to the stabilization time.

Theorem 22. *On an array of processes, the expected number of moves needed to stabilize from a failure of k contiguous nodes is $O(k.n)$.*

Proof. On an array of processes numbered 0 through $n-1$ from left to right, assume that initially $\forall i : v(i) = 1$ holds. Let a failure change the values of $v(0)$ through $v(k-1)$ $(1 < k < n)$ to 0. Whenever the node to the left of the boundary (between 0 and 1) executes move to update its v, the boundary moves one place to the left. Similarly, when the node to the right of the boundary makes a move to update its v, the boundary moves to the right. The system will stabilize when all v become identical.

The probability of the boundary moving to the right (or to the left) is $2^{-\frac{M-1}{2}}$, and the balance reflects the probability of the boundary remaining unchanged. We reduce this computation to a version of gambler's ruin where to win or lose a dollar

(call it a *step*), not one, but $\frac{M-1}{2}$ consecutive heads or tails of a fair coin will be necessary[2]. It follows from [32] that, the expected number of *steps* needed to finish the game is $k \times (n-k)$. Since each *step* here costs an expected number of $2^{\frac{M-1}{2}}$ moves, the expected number of moves needed to finish the game is $2^{\frac{M-1}{2}}.k.(n-k)$. □

To validate this result on a general topology, we need to analyze gambler's ruin in multiple dimensions. We have not done it, but leave this as a conjecture. Based on the fact that a legal configuration is reachable from any initial configuration, the following theorem holds.

Theorem 23. *Algorithm 5.2 is weakly stabilizing.*

5.7 Experimental Results

We ran simulation experiments to study the convergence property for graphs with various degrees and with different values of m. For our experiments, we used random graphs with 1000 nodes. For the first set of experiments presented in Figure 5.3, we set the edge probability value (denoted by p in the graphs) of the random graphs to 0.5.

In the first set of experiments, we varied the value of m in powers of 2. The initial values of x were randomly set. After a single fault was injected, the fraction of cases where the fault was successfully contained was plotted. Because of the probabilistic nature of our algorithm, there were instances when the system of nodes did not

[2]One extra step will be needed whenever the fortune changes from one player to the other, but with large M, we ignore the impact of this, and map the computation to a Markov process.

converge. We set an upper bound (10,000 moves) for the number of moves it required for the system to converge. If convergence had not occurred within this bound, then those instances were treated as failed cases of non-convergence. For each value of m, we ran the experiments 1000 times and plotted the number of moves required for convergence (X-axis) against fraction of converged instances (Y-axis). The empirical distribution function is shown in Figure 5.3.

The larger the value of m, the greater is the difficulty for the fault to contaminate beyond its immediate neighborhood. With increasing m, the containment in space is tighter, but stabilization from arbitrary initial configurations slows down. A lower value of m allows faster recovery from single failures at the expense of an increase in the contamination number. This is consistent with the analysis – the value of m represents the effectiveness of the "fence" around the faulty zone. Also, the *stabilization time* increases for smaller values of m, since the fault contaminates a small number of nodes beyond the distance-1 neighbors, and recovery from multiple failures is inefficient. These properties are evident from our graph.

It is to be noted that for $m = 2$ and for $m = 32$, the two extreme m values we had chosen in our experiments, the system exhibits the expected behavior. But for middle values of m, some anomaly can be observed. For example, $m = 16$, rises over $m = 8$ and $m = 4$. This shows the potential for using m as the tuning parameter in our experiments. There is a threshold value of m, beyond which the system may be in a "stuttering mode" since neither the fault propagates beyond the distance-1 neighbors, nor the recovery is complete. In our case, between $m = 4$ and $m = 8$, this

phenomenon was observed. Therefore fraction of converged instances for those values lag behind.

In the second set of experiments, we set $m = 2$. Again the initial values of x were randomly set. After a single fault was injected, the fraction of cases where the fault was successfully contained was plotted. Because of the probabilistic nature of our algorithm, there were instances when the system of nodes did not converge. We set an upper bound ($10,000$ moves) for the number of moves it required for the system to converge. If convergence had not occurred within this bound, then those instances were treated as failed cases of non-convergence. This time we varied p, which denotes the edge probability for the random graph. For each value of p, we ran the experiments 1000 times and plotted the number of moves required for convergence (X-axis) against fraction of converged instances (Y-axis). The empirical distribution function is shown in Figure 5.4.

Our second experiment captures the fact how node degrees can influence the convergence of the system. For a Δ-ary tree, this is equivalent to varying the value of Δ while keeping m fixed. As we used random graphs for our experiments, we could not directly vary Δ. Rather we chose different values of p to control the sparseness/density of the graphs. As expected, for dense graphs (e.g., $p = 0.8$), the fraction of converged instances was low.

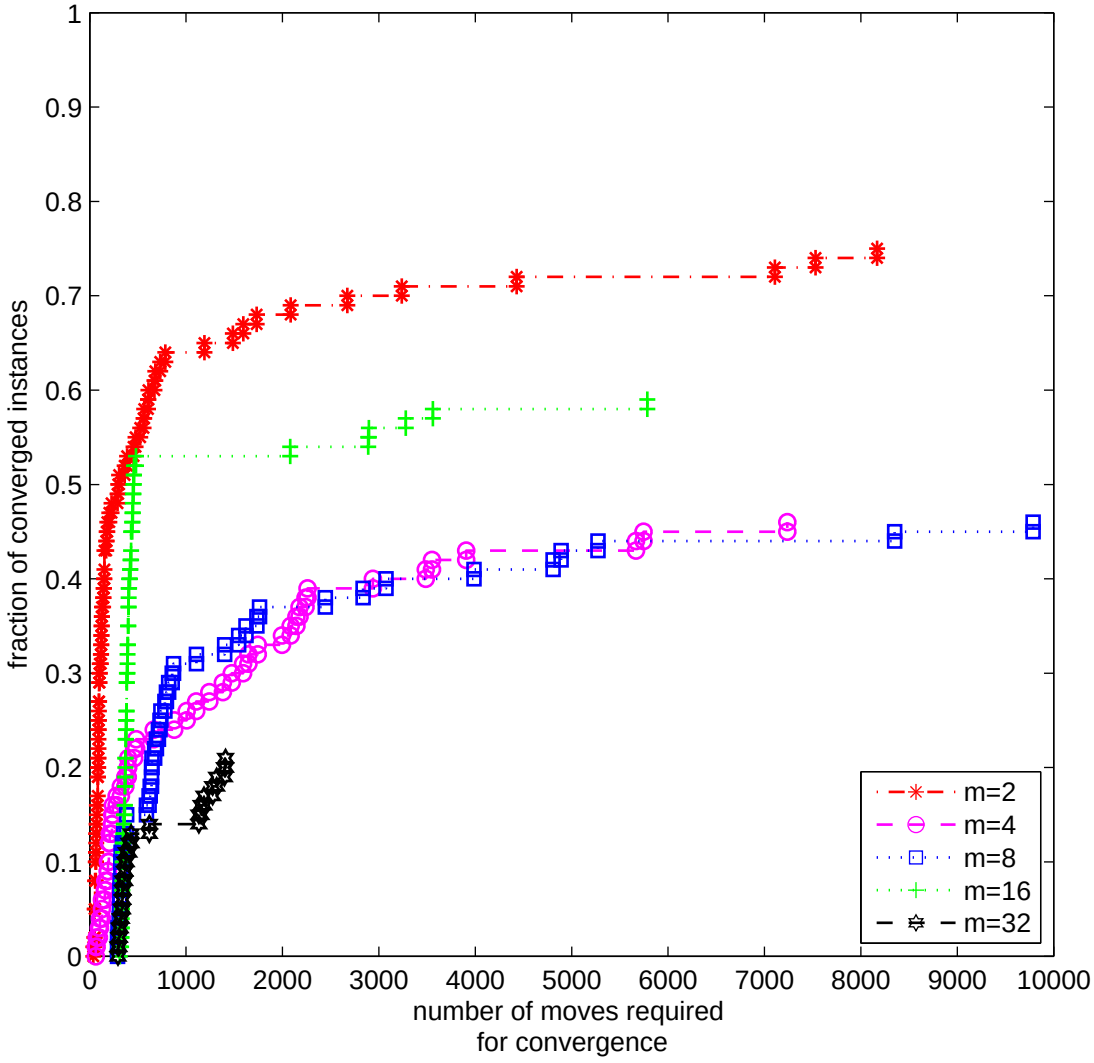

Figure 5.3: Simulation results for various values of m with p as a parameter, where p is the edge probability of the random graph.

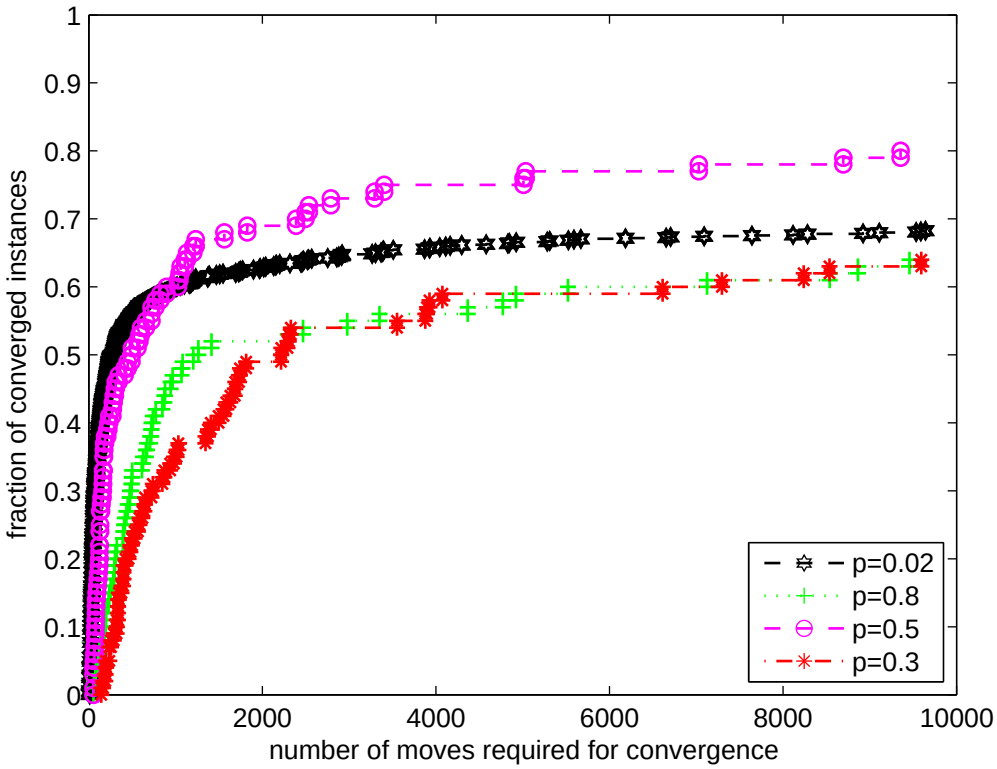

Figure 5.4: Simulation results for various values of p with m as a parameter.

5.8 Discussions

The proposed algorithms allow the immediate neighbors of the faulty process to be contaminated. However as m increases, it becomes increasingly difficult for the failure to propagate to the distance-2 neighbors and beyond. However, a high value of m increases the stabilization time. One can use m (or M for the bounded version of the solution) as a tuning parameter to tune the performance of the protocol. A lower value of m allows faster recovery at the expense of an increase in the contamination number, and the recovery time from multiple failures also improves.

The major advantages of the proposed technique is that the fault-gap is independent of the network size. This increases the availability of the system by restoring the system's readiness to efficiently tolerate the next single fault within a short time. This is where our algorithm is different from other algorithms, where the fault-gap is $O(n)$ or worse.

As the degree of the nodes increases, the containment time increases fairly rapidly. Therefore the algorithm is suitable for sparse topologies. When the topology is dense or the clustering coefficient is large, although the containment time increases, a smaller fraction of the distance-1 neighbors is contaminated.

The solution to the persistent-bit problem can be utilized to find fault-containing algorithms for more general problems, one of them is presented in the next chapter.

CHAPTER 6
FAULT-CONTAINMENT IN WEAKLY-STABILIZING SYSTEMS

6.1 Introduction

Chapter 5 introduced a new technique for adding fault-containment to a stabilizing system by biasing the impact of the random scheduler. In this chapter, we extend the approach by solving a non-trivial problem in stabilization: the problem of *leader election*. The purpose is to establish that the proposed technique is indeed useful enough to solve a wider class of problems in distributed systems.

We start from the leader election algorithm proposed by Devismes et al. in [14]. The algorithm works on a tree of anonymous processes, and is weakly stabilizing. Failures, however minor they are, put the system in an illegal configuration, from which there is no guarantee for recovery under a deterministic scheduler. Even with a randomized scheduler, the expected recovery time will depend on the size of the network. In this chapter, we transform the algorithm in [14] for an array of anonymous processes, and add fault-containment to it. The end result is that the containment time is $O(1)$, and the contamination number is 4 w.h.p, the bound is attained by setting m to infinity, where m is a user defined tuning parameter.

We follow the general computational model introduced in Chapter 2. In addition, we introduce two additional variables P and x for each process. Each process i has a primary variable $P(i) \in N(i)U\perp$. If $j = P(i)$, then j is called the parent of

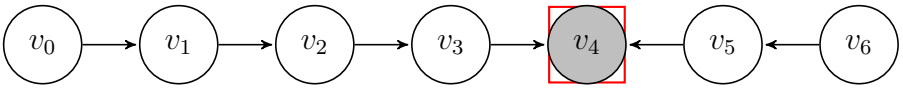

Figure 6.1: A legal configuration with v_4 as the leader.

node i. In addition to P[1], there is a secondary variable $x \in Z+$ that will used for containing the effect of a single failure. The system is in a legal configuration, when the following two conditions hold:

1. For exactly one process i: $P(i) = \perp$
2. for all $j \neq i$, $P(i) = j$ implies $P(j) \neq i$

Here, we use the proceed ids for the sake of identification only. Being an anonymous system, these ids are not used for any kind of decision-making. Figure 6.1 shows a system of processes in its legal configuration, where an arrow from i to j implies that j is the parent of i and a node without a parent pointer implies that its parent is $\perp$. As in Chapter 5, the secondary variable x will reflect the priority of process i in executing some of its actions to update its parent $P(i)$.

Arbitrary configurations may be caused by transient failures that can corrupt the system state. Due to the weakening of the stabilization property, it is not possible to bound the stabilization time under a deterministic scheduler, and randomized scheduling becomes necessary to guarantee that the system recovers to a legal configuration with probability 1.

[1]we use $P(i)$ to denote the parent of node i in fault-containment whereas in selfish stabilization $\pi(i)$ denoted the parent of i

6.2 Contributions

Using a randomized scheduler, we present a solution for the leader election problem on a line topology that is both weakly stabilizing and fault-containing. Our algorithm confines the effect of any single fault to the constant-distance neighborhood of the faulty process with high probability (w.h.p.[2]). More specifically, our algorithm restricts the contamination number to 4 w.h.p., thus the algorithm is spatially fault-containing. The expected recovery time from a single fault is independent of the array size, i.e., the solution is fault-containing in time too.

6.3 Probabilistic Algorithm for Fault-Containment

Our starting point is the weakly stabilizing leader election algorithm on a tree network presented in [14]. For implementing it on an array, we make minor modifications for the stabilization rules. An array is a special case of a tree where each node except the end nodes has a degree of 2. Therefore we replace the notations of Δ from [14], and just consider the two neighbors (or the only neighbor if it is an end node) of a particular process. For the fault-containment part though, we need to add new rules. The basic idea is the same as presented in Chapter 5. To make the protocol fault-containing, we add to each process i a secondary variable $x(i)$ whose domain is the set of non-negative integers. In a way, $x(i)$ will reflect the priority of process i in executing an action to update $P(i)$. Process i will update $P(i)$ and

[2]An event e happens with high probability (w.h.p) if $\lim_{m\to\infty} \Pr(e) = 1$, where m is a user defined parameter. This is slightly different from the traditional definition of w.h.p. [45] in randomized algorithms, but used in the same spirit.

increase its $x(i)$ value with respect to its neighbors when the following conditions hold:

1. The randomized scheduler chooses i,
2. $\{(\exists j \in N(i) : P(i) = j)\}$,
3. $\{(\exists k \in N(i) : P(k) = l \neq i)\}$,
4. $\{x(i) \geq x(k)\}$.

After updating $P(i)$, process i will increase its $x(i)$ value accordingly: $\{x(i) \leftarrow \max_{q \in N(i)} x(q) + m\}$, $m \in \mathbb{Z}^+$ (m is a user defined parameter).

If the first three conditions hold, but not the fourth one, process i increases its $x(i)$ value by 1, but leaves $P(i)$ unchanged. The same thing happens when the first two conditions hold, but not the fourth, and the third condition is modified to $\exists k \in N(i) : P(k) = \perp$.

Observe that once a process i updates $x(i)$, it becomes difficult for its neighbors to change their P-values, since their x-values will lag behind that of i. The larger is the value of m, the greater is the difficulty. A neighbor j of i will be able to update $P(j)$ if it is chosen by the random scheduler m times, without choosing i even once (except case R5 of Algorithm 1, where the update happens immediately when recovery is in sight within a single future move). On the other hand, after making a move, it becomes easier for i to update $P(i)$ again in the near future. With a large value of m, the probability of j being able to change its parent pointer compared to

i is very low. This explains the mechanism of containment. The complete algorithm is described below for a process i:

1. R1 describes the situation when a process i has a parent, but all its neighbors consider i as their parent. So i sets its parent pointer to null and start considering itself as the leader.

2. R2 describes the situation when a process i has no parent and one of its neighbors q does not satisfy the condition $P(q) = i$. Note that for a single-fault scenario it cannot happen that both of i's neighbors do not satisfy the same condition. This means i is not unanimously selected as the leader by its neighbors. As a consequence, i stops considering itself as a leader by setting its parent pointer to q, i.e., $P(i) = q$.

3. R3 a) describes the situation when parent of i is j, and i has a neighbor k whose parent is a node l. Node l is at distance-2 from i. Now if $x(i) \geq x(k)$, then i sets k as its new parent and increases its $x(i)$ value with respect to its neighbors.

4. R3 b) describes the the situation when parent of i is j, j has a parent, and i has a neighbor k whose parent is a node l. Node l is at distance-2 from i. Now if $x(i) < x(k)$, then i does not alter its parent pointer but it just increments its x value by 1.

5. R4 a) describes the situation when parent of i is j, and i has a neighbor k whose parent pointer is set to null. Now if $x(i) \geq x(k)$, then i sets k as its new parent.

Algorithm 6.1 *containment: Program for process i*

- Variable: $P(i) \in N(i) \cup \{\perp\}$.
- Macro: $C(i) = \{q \in N(i) | P(q) = i\}$
- Predicates: $Leader(i) \equiv (P(i) = \perp) \wedge (\forall j \in N(i) : P(j) = i)$
- Actions:

R1 $(P(i) \neq \perp) \wedge (|C(i)| = |N(i)|) \longrightarrow P(i) \leftarrow \perp$

R2 $(P(i) = \perp) \wedge (|C(i)| < |N(i)|) \longrightarrow P(i) \leftarrow (N(i) \setminus C(i))$

R3 a) $(\exists j \in N(i) : P(i) = j) \wedge (P(j) \neq \perp) \wedge$
$(\exists k \in N(i) : P(k) = l \neq i \text{ or } \perp) \wedge (x(i) \geq x(k)) \longrightarrow (P(i) \leftarrow k) \wedge$
$\left(x(i) \leftarrow \max_{q \in N(i)} x(q) + m\right), m \in \mathbb{N}$

b) $(\exists j \in N(i) : P(i) = j) \wedge (P(j) \neq \perp) \wedge (\exists k \in N(i) : P(k) = l \neq i) \wedge$
$(x(i) < x(k)) \longrightarrow x(i) \leftarrow x(i) + 1$

R4 a) $(\exists j \in N(i) : P(i) = j) \wedge (\exists k \in N(i) : P(k) = \perp) \wedge (x(i) \geq x(k)) \longrightarrow$
$P(i) \leftarrow k$

b) $(\exists j \in N(i) : P(i) = j) \wedge (\exists k \in N(i) : P(k) = \perp) \wedge (x(i) < x(k)) \longrightarrow$
$x(i) \leftarrow x(i) + 1$

R5 $(\exists j \in N(i) : P(i) = j) \wedge (P(j) = \perp) \wedge (\exists k \in N(i) : P(k) = l \neq i \text{ or } \perp) \longrightarrow$
$P(i) \leftarrow k$

6. R4 b) describes the situation when parent of i is j, and i has a neighbor k whose parent is set to null. Now if $x(i) < x(k)$, then i does not alter its parent pointer but it just increments its x value by 1.

7. R5 is necessary for recovery. The intuition is if a node finds out that its change of parent will help the system to recover in a single future move, then it makes the move. When parent of i is j, and j has no parent, and there is a neighbor k of i such that k whose parent is a node l. Node l is at distance-2 from i. Now regardless of the value of $x(i)$ i sets k as its new parent.

6.4 Recovery

In this section, we describe the different cases of single-fault configuration and the recovery steps. It is to be noted that we have to consider cases up to distance-4 from the leader, as beyond distance-4, all cases of recovery are similar to each other for single-fault configuration. For convenience, we consider an array of length n and denote process i on the array as v_i where $i = 0, \cdots, n-1$. In each figure; the gray node denotes the original leader in the system, the node with a square is the node hit by the single fault, and the nodes with a $\star$ above are the nodes whose guards are true after the single fault hits the system. If there is an arrow from i to j, that indicates $P(i) = j$.

6.4.1 Fault at the leader

In Fig. 6.2, v_4 is the leader where the fault hits, and v_3 becomes the parent of v_4. In this case, the system recovers trivially in a single step. If the scheduler chooses

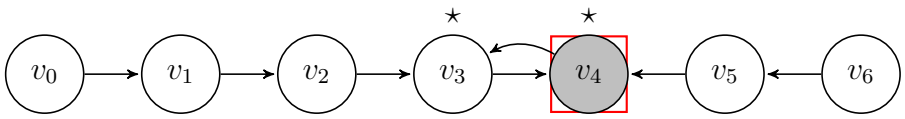

Figure 6.2: Fault at leader v_4. Due to the fault v_3 becomes v_4's parent.

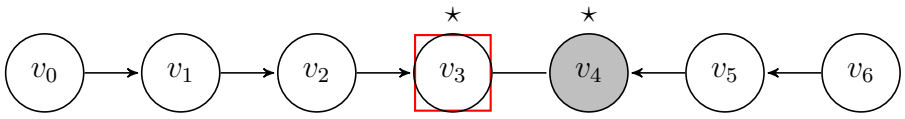

Figure 6.3: Fault at distance-1 neighbor from the leader. v_3's parent pointer becomes null due to the fault.

either of v_3 or v_4, R1 can be applied at v_3 or v_4. Note that, if v_4 makes the move, the system goes back to the initial legal configuration, whereas if v_3 makes the move, v_3 becomes the new leader of the system after recovery.

6.4.2 Fault at distance-1 neighbor from the leader

a) The parent pointer of distance-1 neighbor from the leader becomes null. The fault hits v_3 (Fig 6.3). If v_3's parent pointer becomes null, again the recovery happens trivially in a single step. If the scheduler chooses either of v_3 or v_4, R2 can be applied at v_3 or v_4 and a legal configuration can be reached in a single step. Note that, if v_3 makes the move, the system goes back to the initial legal configuration, whereas if v_4

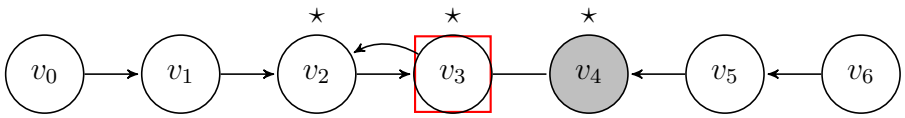

Figure 6.4: Fault at distance-1 neighbor from the leader. Due to the fault v_2 becomes v_3's new parent.

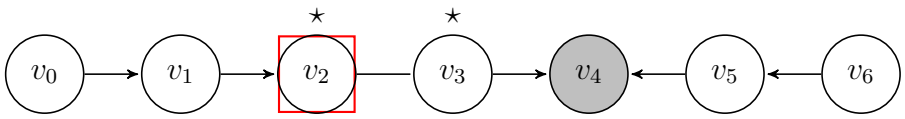

Figure 6.5: Fault at distance-2 from the leader. v_2's parent pointer becomes null due to the fault.

makes the move, v_3 becomes the new leader.

b) A new node becomes the parent of the distance-1 faulty node. In Fig 6.4, if $P(v_3) = v_2$, then v_2, v_3, and v_4's guards are true. If v_3 is selected first, and if $x(v_3) \geq x(v_4)$, the system trivially recovers in a single step (by R4a at v_3). Otherwise, v_3 would not make a parent change. If the scheduler chooses v_4, then the recovery happens in two steps. First, v_4 applies R2 and after that either v_2 or v_3 makes a move (R1). Note that if v_3 makes the move, the leader shifts one place compared to the original legal configuration, whereas if v_2 makes the move, the leader shifts two places.

If v_2 makes the first move, R1 can be applied at v_2. After that if the scheduler chooses v_4, recovery is immediate as R2 can be applied at v_4. But if the scheduler chooses v_3 repeatedly and $x(v_3) \geq x(v_4)$ and $x(v_3) \geq x(v_2)$, oscillations can occur in the system for some period of time due to the fact that R4a is applicable at v_3. v_2 and v_4 alternately becomes v_3's parent. But whenever v_2 or v_4 is chosen by the scheduler next, the system recovers (R2). In the case, $x(v_3) < x(v_2)$ or $x(v_3) < x(v_4)$, recovery is complete when the scheduler next chooses v_2 or v_4 respectively.

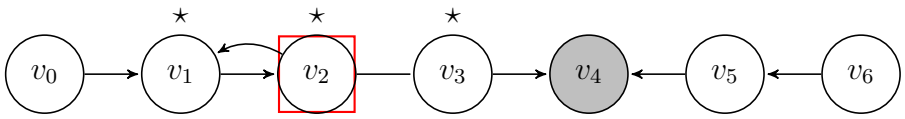

Figure 6.6: Fault at distance-2 from the leader. v_1 becomes the new parent of the v_2 due to the fault.

6.4.3 Fault at distance-2 from the leader

a) The parent pointer of distance-2 neighbor from the leader becomes null. Consider Fig 6.5. The fault hits the system at v_2. If the scheduler chooses v_2, the system trivially recovers in a single step (R2). If v_3 is chosen by the scheduler, the system has a potential for oscillations, i.e., v_4 or v_2 alternately may become v_3's parent, depending on the x values of v_2, v_3, v_4. The system recovers by applying R2 at v_2 or v_4 respectively when either of them is chosen next by the scheduler.

b) A new node becomes the parent of the distance-2 faulty node. Consider Fig 6.6. If v_1 becomes the new parent of v_2, then either v_1, v_2 or v_3 can make a move. If v_2 is selected first, the situation is like the one described in 4.2b), except that R3 a) can be applied now at v_2 as v_3 has a parent if $x(v_2) \geq x(v_3)$. If v_3 is selected first, v_3 can select v_2 as its new parent if $x(v_3) \geq x(v_2)$. The recovery is complete following v_2, v_1's (or v_1, v_2's) moves (R1) followed by v_4's move (R2) or vice versa. If both v_3 and v_2 are unable to change their parents due to smaller x values, then recovery is complete by v_1's move first (R1), followed by v_3's move (now R5 can be applied at v_3) and v_4's move (R2).

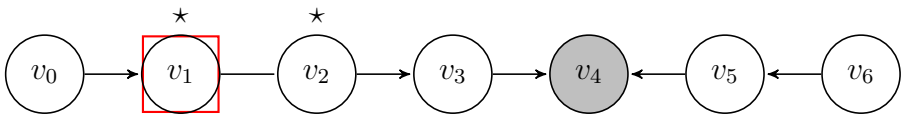

Figure 6.7: Fault at distance-3 from the leader. v_1's parent pointer becomes null due to the fault.

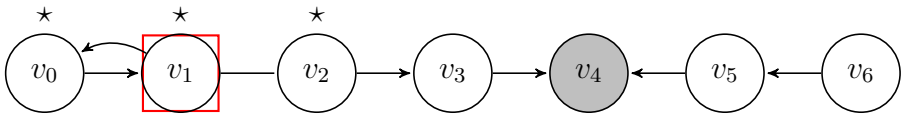

Figure 6.8: Fault at distance-3 from the leader. v_0 becomes the new parent of v_1.

6.4.4 Fault at distance-3 neighbor from the leader

a) The parent pointer of distance-3 neighbor from the leader becomes null. Consider Fig 6.7. If the scheduler chooses v_1, recovery trivially happens in a single step (R2). If v_2 is chosen first, then it can alternately choose v_1 or v_3 as its parent for a while, if its x value is greater than that of both v_1 and v_3. Recovery completes when v_1 is selected (R2). Note that, because of higher x value of v_2, v_3 is unlikely to change its parent. If it does, then recovery happens by applying R5 at v_3 followed by R2 at v_4.

b) A new node becomes the parent of the distance-3 faulty node. Consider Fig 6.8. If v_0 becomes the new parent of v_1, then v_0, v_1, and v_2's guards are true. If v_1 is selected first, then the situation is the same as described above in 4.3b). If v_0 is chosen by the scheduler first, R1 can be applied at v_0. The system recovers when v_1 chooses v_2 as its parent afterwards (using R3a if applicable) and v_0 makes a move next (R2 can be applied at v_0 possibly after some oscillations of v_1). If v_2 is selected to make a

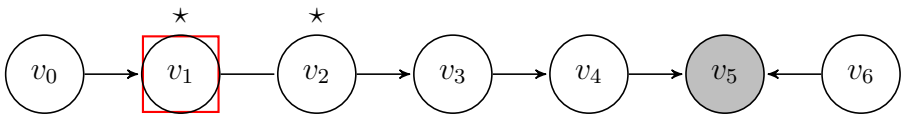

Figure 6.9: Fault at distance-4 from the leader. v_1's parent pointer becomes null.

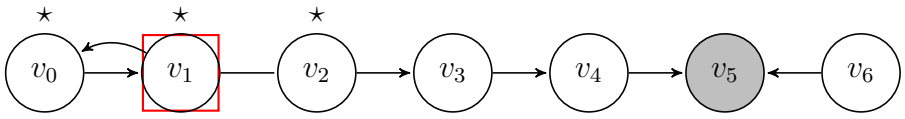

Figure 6.10: Fault at distance-4 from the leader. v_0 becomes the new parent of v_1.

move and if $x(v_2) \geq x(v_1)$, then v_2 chooses v_1 as its parent and increases its x value (R3a). Now v_3 is still able to change its parent to v_2 regardless of the value of x at v_2 (R5) provided v_0 had made a prior move. Recovery is completed by v_4's move (R2). Otherwise, if v_0 had not made a move prior to v_3, the higher x value of v_2 will prevent v_3 to change its parent. Then the recovery is completed by v_1's move (R5) followed by v_0's move (R2). Note that even if v_0 is not an end node, this still applies.

6.4.5 Fault at distance-4 neighbor or beyond from the leader

a) The parent pointer of distance-4 neighbor from the leader becomes null. Consider Fig 6.9. If v_1 is selected by the scheduler, recovery occurs immediately by R2 or after some oscillations at v_2. Otherwise, if $x(v_2) \geq x(v_1)$, then v_1 becomes v_2's new parent (R4a). Similarly v_2 may become v_3's new parent, but this time v_3 sets its x value higher (R3a). Recovery is completed by v_4's move (R5) followed by v_5's move (R2). Note that for fault occurring at distance-4 and beyond from the leader, R5 will no more applicable at the distance-1 neighbor from the leader (v_4 in this case).

Therefore, we do not consider hereafter the cases beyond fault at distance-4 beyond the leader.

b) A new node becomes the parent of the distance-4 faulty node. Consider Fig 6.10. If v_0 becomes the new parent of v_1, then v_0, v_1, and v_2's guards are true. If v_1 is selected to make a move first, then the situation is the same as described above in 4.4b). If v_0 is chosen by the scheduler first, R1 can be applied at v_0. The system recovers when v_1 chooses v_2 as its parent afterwards (R3a if applicable) and v_0 makes a move next (R2 can be applied at v_0 possibly after some oscillations of v_1). If v_2 is selected to make a move and if $x(v_2) \geq x(v_1)$, then v_2 chooses v_1 as its parent and increases its x value (R3a). Now v_3 cannot apply R5 anymore and it is unlikely that the fault will propagate beyond v_2. The recovery proceeds through the following steps - v_0 makes a move (R1), after some oscillations at v_2, v_1 chooses v_2 as its parent (R5), and finally v_0 selects v_1 as its parent (R2). Note that even if v_0 is not an end node, this still applies.

Note that all cases of single-fault configuration beyond distance-4 from the leader will not involve any different recovery steps that are already not covered in the previous scenarios. This is because even if we shift the original place of the fault further away from the leader, its neighborhood that is going to be affected by subsequent recovery steps will remain unchanged. In the recovery mechanism, R3b) and R4b) are not shown as we only highlighted the moves where the change of parent pointers occurs leading to the recovery of the system.

6.5 Results

6.5.1 Fault-containment in space

Theorem 24. *As $m \to \infty$, the effect of a single failure is restricted within distance-4 of the faulty process on an array, and the contamination number is 4, i.e., algorithm* containment *is spatially fault-containing.*

To prove the result of spatial containment, we need to find out how far the observable variables change from the faulty node w.h.p. We consider all the subcases of the recovery mechanism.

1. Fault at leader: The fault propagates to at most distance-1.

2. Fault at distance-1 neighbor from the leader:

   (a) Parent pointer of the distance-1 neighbor becomes null: The fault propagates to at most distance-1.

   (b) A new node becomes the parent of the distance-1 faulty node: In the recovery steps, we showed that in Fig 6.4, at most v_2 or v_4's parent might change. Thus, the fault propagates to at most distance-1.

3. Fault at distance-2 neighbor from the leader:

   (a) The parent pointer of the distance-2 neighbor becomes null: Consider Fig 6.5. If v_2 is selected the system recovers immediately. Another possible recovery is through the sequence of moves of v_3 followed by v_4. In the latter

case the contamination happens up to distance-2 of the original faulty node.

(b) A new node becomes the parent of the distance-2 faulty node: Consider Fig 6.6. The worst case scenario happens when v_3 makes a move and after that v_4 completes the recovery. Contamination happens up to distance-2 of the original faulty node in this case.

4. Fault at distance-3 neighbor from the leader:

(a) The parent pointer of the distance-3 neighbor becomes null: Consider Fig 6.7. The worst case scenario happens when v_4 has to change its parent. The fault propagates to at most distance-3.

(b) A new node becomes the parent of the distance-3 faulty node: Consider Fig 6.8. The worst case scenario happens when v_4 has to change its parent. The fault propagates to at most distance-3.

5. Fault at distance-4 from the leader:

(a) The parent pointer of the distance-4 neighbor becomes null: This is the scenario where the highest spatial contamination occurs. Consider Fig 6.9. In the worst case, a distance-4 node from the faulty node might have to change its parent pointer. In Fig 6.9, v_5 is this node.

(b) A new node becomes the parent of the distance-4 faulty node: Consider Fig 6.10. The fault propagates up to distance-1 w.h.p. The probability

of the fault contaminating beyond distance-1 is $(1 - 1/2^m) \times 1/2^m$ and $\lim_{m\to\infty}(1 - 1/2^m) \times 1/2^m = 0$.

Note that for fault occurring at distance-4 and beyond from the leader, R5 will no more be applicable to the distance-1 neighbor from the leader. Hence algorithm *containment* is spatially fault-containing and the highest contamination number is 4 w.h.p. □

6.5.2 Fault-containment in time

Theorem 25. *The expected number of steps needed to contain a single fault is independent of n, i.e., the number of nodes in the array. Hence algorithm* containment *is fault-containing in time.*

Proof. To prove that algorithm *containment* is fault-containing in time, we again consider each individual subcase of fault. We show that each subcase can be bounded and each of them is independent of n, i.e., the size of the array. For each individual case, we calculate the expected number of moves required for recovery. Essentially that means we are considering the probabilities of the system recovering in a single move, in two moves, in three moves etc. We denote the number of moves (which is a random variable) as X, and $\Pr(X = x)$ denotes the probability that the system recovers using x moves.

1. Fault at leader: Consider Fig. 6.2. This is a trivial case. In this scenario, both v_4 and v_3 have their guards true. Each of them has an equal probability to execute, given a chance. The system recovers in a single move if either of them executes.

So the expected number of moves required for recovery is $1 \times 1/2 + 1 \times 1/2 = 1$.

2. Fault at distance-1 from the leader:

   (a) The parent pointer of the faulty node becomes null: Consider Fig. 6.3. This is again a trivial case. The system recovers in a single move if either v_4 or v_3 executes. The expected number of moves for recovery is the same as before, i.e., $1 \times 1/2 + 1 \times 1/2 = 1$.

   (b) A new node becomes parent of the faulty node: The system can recover following different sequences:

   - v_4, v_2 or v_4, v_3.
   - $v_3, \cdots, v_3$ ($x(4) - x(3)$ times)
   - $v_3, \cdots, v_3$ ($x(4) - x(3) - 1$ times) followed by v_4, v_2 or v_4, v_3
   - v_2, v_4
   - $v_2, v_3, \cdots, v_3$ followed by v_2 or v_4.

   The length of recovery sequences of the first four situations is finite and independent of n, and only the last sequence may be arbitrarily long. We show that its expectation is finite. After v_2 makes a move applying R2, both of v_3's neighbor consider themselves as the leader. Hence depending on the parent of v_3, each time there are at most two enabled nodes: v_2 and v_3 or v_3 and v_4. Therefore, the probability that the scheduler chooses v_3 is $1/2$ before recovery completes. Hence, the probability of v_3 being selected consecutively n times is $1/2^n$. Therefore, if v_2 is selected by the scheduler

first, the expected length of the recovery sequence is $2 + \sum_{n=1}^{\infty} n/2^n = 4$.

3. Fault at distance-2 from the leader

   (a) The parent pointer of the faulty node becomes null: Regardless of the values of $x(2)$, $x(3)$ and $x(4)$, no matter which node the scheduler chooses, there is always $1/2$ probability that the system recovers after the selected node makes a move. Hence, the expected recovery time is $\sum_{n=1}^{\infty} n/2^n = 2$.

   (b) A new node becomes parent of the faulty node: In our proof so far, we assumed the value of x to be different. Proceeding in the same manner, the rest of the expectation calculation can be done. However, for subsequent cases, the computation will be more complex. The constant factors in the result will still hold, since the time complexity calculation for each subcase will involve products of several terms like $(1 - 1/2^m)$ and $1/2^m$. For the sake of simplicity, we assume the value of x to be identical for the rest of the proof.

   Following the above assumption, the expected number of moves for recovery will be:

$$\begin{aligned}\mathbb{E}(X) &= 1 \times \frac{1}{3} + 3\left(\frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{4} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{4} \times \frac{1}{2}\right) \\ &\quad + \frac{2}{3}\sum_{n=2}^{\infty} \frac{2n+1}{2^{2n}} \\ &= \frac{151}{108}\end{aligned}$$

4. Fault at distance-3 from the leader:

(a) Parent pointer of the faulty node becomes null: In this case, the expected number of moves for recovery will be

$$\mathbb{E}(X) = 1 \times \frac{1}{2} + 3 \times \left(\frac{1}{2^3} + \frac{1}{2^3}\right) + 5 \times \left(4 \times \frac{1}{2^5}\right) + \cdots$$
$$= \frac{1}{2} + \sum_{n=1}^{\infty}(2n+1) \times 2n \frac{1}{2^{2n+1}} = \frac{131}{54}$$

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be:

$$\mathbb{E}(X) = 1 \times \frac{1}{3} + 2 \times 0 + 4 \times \left(\frac{1}{3}\frac{1}{2}\left(1 - \frac{1}{2^m}\right)\frac{1}{2^m}\frac{1}{2}\right) + 3 \times \left(\frac{1}{3}\frac{1}{2}\frac{1}{2}\frac{1}{2}\right) + 5 \times \left(\frac{1}{3}\frac{1}{2^4}\right) + \cdots$$
$$\leq \frac{1}{3} + \frac{25}{9} + 4 \times \frac{1}{3}\frac{1}{2}\frac{1}{4}\frac{1}{2} = \frac{115}{36}$$

(The term $\left(1 - \frac{1}{2^m}\right)\frac{1}{2^m}$ is bounded by 1/4).

5. Fault at distance-4 from the leader:

(a) The parent pointer becomes null: In this case, the expected number of moves for recovery will be: $\mathbb{E}(X) = \frac{1}{2} + \sum_{n=1}^{\infty}(2n+1)\frac{1}{2^{2n+1}} = 10/9$.

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be: $\mathbb{E}(X) = 1 \times \frac{1}{3} + 2 \times 0 + 3 \times \left(\frac{1}{3}\frac{1}{2}\frac{1}{2}\right) + 4 \times \frac{1}{3}\frac{1}{4}\frac{1}{2}\frac{1}{2} + 5 \times \frac{1}{3}\frac{1}{2^4} + 7 \times \frac{1}{3}\frac{1}{2^6} + \cdots = 29/27$

6. Fault beyond distance-4 from the leader:

(a) The parent pointer becomes null: In this case, the expected number of moves for recovery will be: $\mathbb{E}(X) = \frac{1}{2} + 3 \times \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) + 5 \times \left(4 \times \frac{1}{2^5}\right) = \frac{33}{32}$.

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be:

$$\mathbb{E}(X) = 1 \times \frac{1}{3} + 2 \times 0 + 4 \times \left(\frac{1}{3}\frac{1}{2}\left(1 - \frac{1}{2^m}\right)\frac{1}{2^m}\frac{1}{2}\right) + 3 \times \left(\frac{1}{3}\frac{1}{2}\frac{1}{2}\frac{1}{2}\right) + 5 \times \left(\frac{1}{3}\frac{1}{2^4}\right) + \cdots$$
$$\leq \frac{1}{3} + \frac{25}{9} + 4 \times \frac{1}{3}\frac{1}{2}\frac{1}{4}\frac{1}{2} = \frac{115}{36}$$

In all individual subcases, the calculation shows that the recovery happens in a finite number of moves and it is independent of n, where n is the size of the array. The maximum expected number of moves required for recovery is 115/36, when a fault occurs at distance-3 from the leader and a new node becomes parent of the faulty node, or when a fault occurs at more than distance-4 from the leader and a new node becomes parent of the faulty node . Therefore algorithm *containment* is fault containing in time. □

6.6 Computing The Availability

An interesting aspect of the proposed algorithm is that there is no overhead for stabilizing the secondary variables. $LC_s = true$ as long as $x(i) \in \mathbb{Z}^+$. So, LC holds as soon as LC_p holds. This leads to the following theorem:

Theorem 26. *For single failures, the fault-gap equals the containment time.*

6.6.1 Convergence

We use martingale convergence theorem to prove the convergence of algorithm *containment*. We first give the definition of martingales, and then we provide the statement of martingale convergence theorem with a corollary derived by the martingale convergence theorem [45]. Finally we show that the corollary can be applied in

our problem.

Definition 12. *Let $\mathcal{F}_n$ be an increasing sequence of σ-fields, and let $X_n \in \mathcal{F}_n$ for all n. X is said to be a martingale with respect to $\mathcal{F}_n$ if the following conditions hold:*

1. $\mathbb{E}(|X_n|) < \infty$,
2. *X_n is adapted to $\mathcal{F}_n$,*
3. $\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$ *for all n*

If $=$ in the last condition is replaced by $\leq$ or $\geq$, then X_n is said to be a supermartingale or submartingale, respectively.

Theorem 27 (The martingale convergence theorem)**.** *If X_n is a submartingale with $\sup \mathbb{E}(X_n^+) < \infty$ then as $n \to \infty$, X_n converges almost surely (i.e. the probability that X_n converges is 1) to a limit X with $\mathbb{E}(|X|) < \infty$.*

Corollary 1. *If $X_n \geq 0$ is a supermartingale then as $n \to \infty$, $X_n \to X$ almost surely and $\mathbb{E}(X) \leq \mathbb{E}(X_0)$.*

Theorem 28. *Algorithm* containment *is weakly-stabilizing.*

Proof. We denote the number of nodes whose guards are true at step i by X_i. Let $\mathcal{F}_i = \sigma < X_0, X_1, \cdots, X_i >$, we first prove that the sequence of X_i is a supmartingale with respect to $\mathcal{F}_i$: (a) $\mathbb{E}(X_i) < \infty$ is trivially true as there are n nodes in the system, so $\mathbb{E}(X_i) \leq n$. (b) $X_i \in \mathcal{F}_i$ by the definition of $\mathcal{F}_i$; (c) We show that $\mathbb{E}(X_{i+1}|X_i) \leq X_i$ by enumerating all possible values of X_i. X_i can only be 0, 2 or 3. Note that X_i's

value cannot be 1. There is no single-fault configuration for which only one node's guard can be true in the system.

1. When $X_i = 0$, $X_{i+1} = 0$ as the system has reached the non-faulty configuration.

2. When $X_i = 2$, $\mathbb{E}(X_{i+1}|X_i) = 1/2 \times 0 + 1/2 \times 2 = 1 \leq 2$.

3. When $X_i = 3$, $\mathbb{E}(X_{i+1}|X_i) = 1/3 \times 0 + 1/3 \times 2 + 1/3 \times 4 = 2 \leq 3$.

As the number of nodes whose guards are true are nonnegative, we apply corollary 1 and thus $X_n \rightarrow X$, *i.e.*, the number of nodes whose guards are true converge.

□

6.7 Discussions

The proposed algorithm allows up to distance-4 neighbors of the faulty process to be contaminated by a single failure. With high probability, the failure does not propagate beyond that. The major advantage of the proposed technique is that the fault-gap is independent of the array size. The expected number of moves required for recovery is constant. This increases the availability of the system by restoring the system's readiness to efficiently tolerate the next single fault within a short time.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Summary

7.1.1 Selfish Stabilization

Selfish stabilization reduces to classical stabilization when the private goals of the constituent processes do not conflict. The following issues are relevant to the approach taken in this dissertation:

The first is the separation of *cooperation* and *competition*. Assume that processes first cooperate to form a tree, then try to optimize it to improve their individual payoffs. In presence of arbitrary initializations, failures, and selfish motives, such segregation of actions is difficult to implement.

The uniqueness of the equilibrium point is another significant issue. When the system of processes has a unique equilibrium point, it reflects the *Nash equilibrium*. However, in several cases, multiple equilibria configurations are possible, and final configuration will depend on the schedule of actions. If the costs of these configurations are distinct, then it leaves open the possibility that after reaching an inferior equilibrium configuration, an unhappy (or ambitious) process can deliberately introduce a perturbation (by corrupting a local variable) to possibly reach a different equilibrium configuration with a better payoff. This underscores the importance of identifying cases where a single equilibrium exists.

7.1.2 Fault-Containment

Historically *fault-containment* has been studied for (strongly) stabilizing systems, but little attention has been paid to adding fault-containment to weakly stabilizing systems. Our work fills this gap. Randomized scheduling guarantees that the system reaches a legal configuration with probability 1, so the choice of *randomized scheduler* is automatic in the context of containment.

The proposed algorithm of Chapter 5 allows the immediate neighbors of the faulty process to be contaminated. However, with high probability, the failure does not propagate to the distance-2 neighbors and beyond. One can use m (or M for the bounded version of the solution) as a tuning parameter to tune the performance of the protocol. A lower value of m allows faster recovery at the expense of an increase in the *contamination number*. A larger value of m, on the other hand, decreases the *contamination number*, but this comes with an increase in the recovery time from multiple failures. It is up to the user to choose an appropriate value of m in a given scenario.

In the solution of Chapter 6, the unbounded variable $x(i)$ can be bounded using the method described in Chapter 5.The proposed algorithm allows up to distance-4 neighbors of the faulty process to be contaminated. With high probability, the failure does not propagate beyond that. The expected number of moves required for recovery is independent of the size of the network.

The major advantages of the two proposed techniques is that the *fault-gap* is independent of the network size. This increases the availability of the system by

restoring the system's readiness to efficiently tolerate the next single fault within a short time. This is where our algorithms are different from other algorithms, where in some cases, the *fault-gap* is $O(n)$ or worse.

For the persistent bit problem solution, as the degree of the nodes increases, the stabilization time increases fairly rapidly. Therefore the algorithm is suitable for sparse topologies. When the topology is dense or the clustering coefficient is large, although the stabilization time increases, a smaller fraction of the distance-1 neighbors is contaminated.

7.2 Future work

7.2.1 Selfish Stabilization

The stabilizing algorithms of Chapters 3 and 4 are weakly stabilizing solutions. *Weak stabilization* is strictly weaker than (strong) self-stabilization and it is a good approximation of (strong) self-stabilization. But whether there exists a strongly stabilizing solution of these problems remains an open question.

While the problems of *shortest path tree* and *maximum flow tree* investigated here are essentially global in nature, many self-stabilizing protocols exist where the legal configurations are defined by local predicates, such as maximal independent set or maximal matching, and one can come up with selfish versions of them. It is worth investigating whether the results presented in this thesis still hold for those problems, or pose new challenges.

In the context of "inferior" or "superior" equilibria, it is worth exploring how

good our solution is compared with the best solution. it is to be noted that non-compliance to global mandates can have an overall negative impact on the payoffs when the *Nash equilibrium* corresponds to an inferior equilibrium. There are different approaches to handle this problem. Development of a payment scheme to reward compliance is one possible approach, while another approach involves detecting cheaters and appropriately penalizing them to force compliance. Quantification of these issues in the current setting is an open problem, and is a topic of future research. In certain cases, our methods may allow situations where the system does not converge to an equilibrium configuration despite the fact there exists at least one such equilibrium in the system. This is different from the case where there exists no equilibrium in the system. The system may exhibit oscillatory behavior forever as described in Chapter 3. Penalties or rewards can be used in such cases too to force cooperation. The selfishness property for each individual class of nodes is partially sacrificed when co-operation is employed, and from the context of individual payoff of each class, the solution may not be optimal anymore, yet the global goal is accomplished, and the system stabilizes.

Selfishness is not the only challenge in todays distributed systems. Often, modern systems have to cope with malicious adversaries who seek to degrade the utility of the system independently of their own cost. Design of algorithms in the presence of malicious nodes is a challenging issue.

7.2.2 Fault Containment

The algorithm for array in Chapter 6 can easily be extended for tree networks. An array is a special case of a tree network. In case of a general tree topology, one will have to consider all the neighbors of a *process* i when executing the rules of the algorithm, instead of considering at most two neighbors for an array. The analysis will involve Δ, the maximum degree of a node, and instead of expected recovery time being constant, it will involve an expression in terms of Δ. Note that this result will satisfy the definition of *weak fault-containment* described in Chapter 2. How to extend our fault-containing leader election algorithm for general graphs remains an open problem.

For the two problems we addressed, in both cases we handcrafted the solutions for adding fault-containment by biasing the schedule of the random scheduler in different ways. Whether a general transformer can be built for the purpose remains an open problem.

REFERENCES

[1] Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering*, 19(11):1015–1027, 1993.

[2] Joffroy Beauquier, Ajoy Kumar Datta, Maria Gradinariu, and Frédéric Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago J. Theor. Comput. Sci.*, 2002, 2002.

[3] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. Optimal reactive k-stabilization: the case of mutual exclusion (extended abstract). In *In PODC99 proceedings of the nineteenth annual ACM symposium on principles of distributed computing*, pages 209–218, 1999.

[4] Doina Bein and Ajoy K. Datta. A self-stabilizing directed diffusion protocol for sensor networks. *Parallel Processing Workshops, International Conference on*, 0:69–76, 2004.

[5] Sarah Brocklehurst, P. Y. Chan 0002, Bev Littlewood, and John Snell. Recalibrating software reliability models. *IEEE Trans. Software Eng.*, 16(4):458–470, 1990.

[6] Hui Cao, Emre Ertin, Vinodkrishnan Kulathumani, Mukundan Sridharan, and Anish Arora. Differential games in large-scale sensor-actuator networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 77–84, New York, NY, USA, 2006. ACM.

[7] Eddy Caron, Ajoy K. Datta, Franck Petit, and Cedric Tedeschi. Self-stabilization in tree-structured peer-to-peer service discovery systems. *Reliable Distributed Systems, IEEE Symposium on*, 0:207–216, 2008.

[8] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett.*, 39(3):147–151, 1991.

[9] Jorge Arturo Cobb, Mohamed G. Gouda, and Ravi Musunuri. A stabilizing solution to the stable path problem. In *Self-Stabilizing Systems*, pages 169–183, 2003.

[10] Johanne Cohen, Anurag Dasgupta, Sukumar Ghosh, and Sébastien Tixeuil. An exercise in selfish stabilization. *ACM Trans. Auton. Adapt. Syst.*, 3(4):1–12, 2008.

[11] Anurag Dasgupta, Sukumar Ghosh, and Sébastien Tixeuil. Selfish stabilization. In *Stabilization, Safety, and Security of Distributed Systems*, volume 4280/2006 of *Lecture Notes in Computer Science*, pages 231–243. Springer Berlin/Heidelberg, 2006.

[12] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In *Stabilization, Safety, and Security of Distributed Systems*, volume 4838/2007 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin/Heidelberg, 2007.

[13] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Fault-containment in weakly-stabilizing systems. In *Stabilization, Safety, and Security of Distributed Systems*, volume 5873/2009 of *Lecture Notes in Computer Science*, pages 209–223. Springer Berlin/Heidelberg, 2009.

[14] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. *Distributed Computing Systems, International Conference on*, 0:681–688, 2008.

[15] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[16] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. In *Chicago Journal of Theoretical Computer Science*, pages 3.1–3.15, 1995.

[17] Sukumar Ghosh and Arobinda Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Inf. Process. Lett.*, 59(5):281–288, 1996.

[18] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 45–54, New York, NY, USA, 1996. ACM.

[19] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007.

[20] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. Fault-containing network protocols. In *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing*, pages 431–437, New York, NY, USA, 1997. ACM.

[21] MG Gouda and M Schneider. Stabilization of maximal metric trees. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association*

with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems), pages 10–17. IEEE Computer Society, 1999.

[22] Mohamed G. Gouda. The theory of weak stabilization. In *WSS '01: Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 114–123, London, UK, 2001. Springer-Verlag.

[23] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. Policy disputes in path-vector protocols. In *ICNP '99: Proceedings of the Seventh Annual International Conference on Network Protocols*, pages 21–30, Washington, DC, USA, 1999. IEEE Computer Society.

[24] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, 2002.

[25] Arobinda Gupta. *Fault-containment in self-stabilizing distributed systems*. PhD thesis, Department of Computer Science, The University of Iowa, Iowa City, IA, USA, 1997.

[26] Joseph Y. Halpern. A computer scientist looks at game theory. Game theory and information, EconWPA, November 2004.

[27] Ted Herman. Self-stabilization bibliography: Access guide. In *Journal of Theoretical Computer Science, Working Paper WP-1, initiated*, 1998.

[28] Ted Herman. Superstabilizing mutual exclusion. *Distrib. Comput.*, 13(1):1–17, 2000.

[29] Ted Herman. Models of self-stabilization and sensor networks. In *IWDC*, volume 2918/2004 of *Lecture Notes in Computer Science*, pages 205–214. Springer Berlin/Heidelberg, 2003.

[30] Tetz C. Huang. A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity. *J. Comput. Syst. Sci.*, 71(1):70–85, 2005.

[31] Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 131–140, New York, NY, USA, 2009. ACM.

[32] Coolidge J.L. The gambler's ruin. *The Annals of Mathematics*, 10:181–192, 1909.

[33] Idit Keidar, Roie Melamed, and Ariel Orda. Equicast: scalable multicast with selfish users. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 63–71, New York, NY, USA, 2006. ACM.

[34] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theor. Comput. Sci.*, 220(1):93–111, 1999.

[35] Shay Kutten and David Peleg. Fault-local distributed mending. In *In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 20–27, 1994.

[36] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *SIGOPS Oper. Syst. Rev.*, 19(4):34–44, 1985.

[37] Marios Mavronicolas, Vicky G. Papadopoulou, Anna Philippou, and Paul G. Spirakis. A graph-theoretic network security game. In *WINE*, pages 969–978, 2005.

[38] Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. On the topologies formed by selfish peers. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 133–142, New York, NY, USA, 2006. ACM.

[39] Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. When selfish meets evil: byzantine players in a virus inoculation game. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 35–44, New York, NY, USA, 2006. ACM.

[40] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[41] Tim Roughgarden and Eva Tardos. How bad is selfish routing? In *IEEE Symposium on Foundations of Computer Science*, pages 93–102, 2000.

[42] Ayman Shaker, , Ayman Shaker, and Douglas S. Reeves. Self-stabilizing structured ring topology p2p systems. In *th IEEE Int. Conference on Peer-to-Peer Computing, 2005*, pages 39–46. IEEE Computer Society, 2005.

[43] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology p2p systems. *Peer-to-Peer Computing, IEEE International Conference on*, 0:39–46, 2005.

[44] H. Simon. *The Models of Bounded Rationality*, volume 1 and 2. MIT press, 1982.

[45] Jaynes Wayman and E. T. Jaynes. Probability theory as logic. Kluwer Academic Publishers, 1990.

[46] Christoph Weyer, Volker Turau, Andreas Lagemann, and Jrg Nolte. Programming wireless sensor networks in a self-stabilizing style. *International Conference on Sensor Technologies and Applications,*, 0:610–616, 2009.