OUKA
The University of Osaka
Institutional Knowledge Archive

Title	A Study on Hierarchical Design of Fault-containing Self-stabilizing Protocols
Author(s)	山内, 由紀子
Citation	大阪大学, 2009, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/2509
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

A Study on Hierarchical Design of Fault-containing Self-stabilizing Protocols

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2009

Yukiko YAMAUCHI

List of Related Publications

Journal Papers

1. Yukiko Yamauchi, Sayaka Kamei, Fukuhito Ooshita, Yoshiaki Katayama, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Hierarchical composition of self-stabilizing protocols preserving the fault-containment property", *IEICE Transactions on Information and Systems* (to appear).

2. Yukiko Yamauchi, Toshimitsu Masuzawa, and Doina Bein, "Preserving the fault-containment property of ring protocols executed on trees", *The Computer Journal* (to appear).

Conference Papers

3. Yukiko Yamauchi, Doina Bein, and Toshimitsu Masuzawa, "Minimizing the message complexity on embedded protocols", *Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems* (Poster), Detroit, USA, Nov. 2008.

4. Yukiko Yamauchi, Sayaka Kamei, Fukuhito Ooshita, Yoshiaki Katayama, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Timer-based composition of fault-containing self-stabilizing protocols", *Proceedings of the 2nd International Symposium on Intelligent Distributed Computing*, pp.217-226, Catania, Italy, Sep. 2008.

5. Yukiko Yamauchi, Doina Bein, Linda Morales, Toshimitsu Masuzawa, and I. Hal Sudborough, "Calibrating an embedded protocol on an asynchronous system", *Proceedings of the 2nd International Symposium on Intelligent Distributed Computing*, pp.227-236, Catania, Italy, Sep. 2008.

6. Yukiko Yamauchi, Toshimitsu Masuzawa, and Doina Bein, "Ring embedding preserving the fault-containment", *Proceedings of the 7th International Conference on Applications and Principles of Information Science*, pp.43-46, Auckland, New Zealand, Jan. 2008.

7. Yukiko Yamauchi, Sayaka Kamei, Fukuhito Ooshita, Yoshiaki Katayama, Hirotsugu Kakugawa and Toshimitsu Masuzawa, "Composition of fault-containing protocols based on recovery waiting fault-containing composition framework", *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pp.516-532, Dallas, USA, Nov. 2006.

Technical Reports

8. Yukiko Yamauchi, Sayaka Kamei, Fukuhito Ooshita, Yoshiaki Katayama, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Timer-based composition technique for self-stabilizing protocols preserving the fault-containment property", *Technical Report of IPSJ*, 2008-AL-118, Vol.2008, No.49, pp.1-8, May 2008.

9. Yukiko Yamauchi, Toshimitsu Masuzawa, and Doina Bein, "Emulation of ring protocols on trees preserving fault-containment", *Technical Report of IEICE*, COMP2006-52, Vol.106, No. 566, pp.13-20, Mar. 2007.

List of Unrelated Publications

Journal Papers

10. Gen Nishikawa, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "A fair self-stabilizing mutual exclusion protocol for mobile ad hoc networks", *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science* (letter), Vol.J91-A, No.02, pp.279-284, Feb. 2008 (in Japanese).

11. Yukiko Yamauchi, Yoshihiro Nakaminami, Fukuhito Ooshita, and Toshimitsh Masuzawa, "TDMA slot assignment for wireless networks based on distance-2 coloring". *Transactions of IPSJ*, Vol.48, No.1, pp.327-341, Jan. 2007 (in Japanese).

Conference Papers

12. Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Output stability of self-stabilizing protocols against topology changes and transient faults", *Proceedings of the 8th International Conference on Applications and Principles of Information Science* (to appear).

13. Yukiko Yamauchi, Takashi Itou, Gen Nishikawa, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Clustering algorithm for mobile ad-hoc networks to improve the stability of clusters", *Proceedings of the IASTED International Conference on Sensor Networks 2008*, pp. 9-15, Crete, Greece, Sep. 2008.

Technical Reports

14. Gen Nishikawa, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "A self-stabilizing mutual exclusion protocol minimizing effect of topology

changes for mobile ad hoc networks", *Technical Report of IPSJ*, 2007-DPS-130, 2007-CSEC-36, pp.177-182, Mar. 2007 (in Japanese).

15. Yukiko Yamauchi, Yoshihiro Nakaminami, Fukuhito Ooshita, Toshimitsu Masuzawa, "TDMA slot assignment for wireless networks based on distance-2 graph coloring", *Technical Report of IPSJ*, 2005-DPS-123, Vol.2005, No.58, pp.69-74, June 2005 (in Japanese).

Abstract

A distributed system consists of processes communicating with each other through communication links. Recently, large scale networks have been developed, e.g. the Internet, ad-hoc networks, sensor networks, inter-vehicle networks. As the number of processes grows, a distributed system is more prone to faults. Fault tolerance is one of the most challenging problems in distributed systems and the design of fault-tolerant distributed protocols attracts more and more attention. The effect of faults may spread over the entire network due to the communication among processes. In large scale networks, it is desired that the effect of a fault is contained and does not contaminate the entire network. In addition, it is expected that the system recover quickly so that the system can tolerate the next fault.

Self-stabilization is one of the most powerful design paradigms for non-masking fault tolerance in distributed systems. A self-stabilizing protocol promises autonomous adaptability against any finite number of transient faults that corrupt memory contents at processes. *Fault-containment* has been attracted much attention in the area of adaptive stabilization. A fault-containing self-stabilizing protocol promises containment of the effect of a small scale fault in addition to self-stabilization against large scale faults. (In the following, we call it fault-containing protocol.) Containment guarantees that the effect of a fault is contained around faulty processes (*spatial containment*) and/or it lasts only a short period of time after the fault (*temporal containment*). The notion of fault-containment is useful in practice. Self-stabilization promises fault tolerance against any finite number of transient faults, but guarantees nothing during the stabilization, e.g. the effect of a small scale fault may spread over the entire system. In practice, catastrophic faults rarely occur while small scale faults are more likely to occur frequently. However, designing fault-containing protocols is generally difficult. The difficulty lies in the fact that it is difficult and costly to detect faulty configuration and faulty processes in distributed settings, while it is necessary for the containment of the effect of faults.

In this dissertation, we focus on the design of fault-containing protocols and propose a framework that facilitates the design of new fault-containing protocols. We propose two methods to realize hierarchical structures of fault-containing protocols that ease the design of new fault-

containing protocols and that improves the reusability of existing protocols by extending their applications.

First of all, we present *fault-containing composition* that provides a hierarchical composition of fault-containing protocols with preserving the fault-containment property of source protocols. In a hierarchical composition of two (or more) protocols, the output of one protocol (the lower protocol) is used as the input to the other protocol (the upper protocol). Though several hierarchical composition techniques for self-stabilizing protocols have been proposed, they cannot preserve the fault-containment property of source protocols. The problem is that existing composition techniques allow the upper protocol to be executed on the incorrect input from the lower protocol and the effect of a fault may spread over the entire network in the upper protocol. Our approach is to control the execution of source protocols so that the upper protocol stops during the recovery of the lower protocol. The difficulty lies in how to guarantee the recovery of the lower protocol when the upper protocol starts its execution. A fault-containing protocol provides temporal containment and/or spatial containment that can be used to guarantee the recovery of the protocol. We propose two types of fault-containing composition methods: to guarantee the recovery of the lower protocol, one utilizes the temporal containment of source protocols (Chapter 3), and the other utilizes the spatial containment of source protocols (Chapter 4). Fault-containing composition is the first step to facilitate the design of fault-tolerant protocols because it shows the possibility of a uniform composition framework for fault-tolerant protocols.

Secondly, we present a simulation technique for fault-containing protocols on an embedded topology. Topology embedding is to embed a virtual topology on a real topology, and this enables a distributed protocol designed for a specific topology to be executed on another topology. A *one-to-one node embedding* is a topology embedding such that one real process corresponds to just one virtual process. Any one-to-one node embedding has natural fault tolerance because when a real process is corrupted by a fault, only one corresponding virtual process is corrupted in the virtual topology. However, one-to-one node embedding introduces dilation (the maximum distance of a virtual link in a real topology) bigger than one and the data on a virtual link may be corrupted by a corruption of intermediate real processes. To preserve the fault-containment of original protocols executed on the embedded topology, it is necessary that the data on a virtual link is not corrupted. As one of the most investigated networks in distributed computing, a ring network is frequently used for distributed computation and control. We focus on ring embedding on a rooted tree and propose a simulation technique for fault-containing ring protocols on an arbitrary rooted tree (Chapter 5). The proposed simulation technique demonstrates the possibility of a universal simulation technique for fault-tolerant protocols executed any embedded topologies.

Contents

1 Introduction **1**

1.1 Distributed Systems . . . 1

1.2 Fault Tolerance of Distributed Systems . . . 2

1.3 Self-stabilization . . . 3

1.3.1 Self-stabilization and Adaptive Stabilization . . . 3

1.3.2 Fault-containment . . . 5

1.4 Hierarchical Design of Distributed Protocols . . . 6

1.4.1 Composition . . . 6

1.4.2 Topology Embedding . . . 7

1.5 Overview of This Dissertation . . . 9

1.5.1 Hierarchical Composition of Fault-containing Protocols . . . 9

1.5.2 Ring Embedding . . . 10

1.6 Organization of This Dissertation . . . 11

2 Preliminary **13**

2.1 Network and Processes . . . 13

2.2 Self-stabilization . . . 14

3 Hierarchical Composition with Temporal Containment **17**

3.1 Fault-containing Composition and RWFC Strategy . . . 19

3.2 Preliminary . . . 21

3.3 Composition Framework . . . 21

3.3.1 Specification of the Local Neighborhood Synchronizer . . . 23

3.3.2 Composition Protocol *RWFC-LNS* . . . 25

3.3.3 Correctness Proof of *RWFC-LNS* . . . 25

3.4 Local Neighborhood Synchronizer . . . 27

3.4.1 Protocol *LNS* . . . 27

3.4.2 Correctness Proof: Stabilization of *LNS* 30
3.4.3 Correctness Proof: Synchronization of *LNS* 36
3.5 Concluding Remarks 37

4 Hierarchical Composition with Spatial Containment 39
4.1 Preliminary 40
4.2 Composition Framework 41
4.2.1 Specification of the Inconsistency Detector 42
4.2.2 Composition Protocol *RWFC-IcD* 43
4.2.3 Correctness Proof of *RWFC-IcD* 44
4.3 Inconsistency Detector 45
4.4 Concluding Remarks 47

5 Ring Embedding Preserving Fault-containment Property 49
5.1 Preliminary 51
5.2 Causal Simulation 53
5.3 Causal Simulation Framework 59
5.3.1 Causal Simulation Protocol RET 61
5.3.2 Correctness Proof of *RET* 62
5.3.3 Performance Evaluation 74
5.4 Example of 1-fault-containing Leader Election 75
5.5 Concluding Remarks 78

6 Conclusion 79
6.1 Summary of the Results 79
6.2 Future Directions 80

List of Figures

1.1 Self-stabilization and fault-containment (Spanning tree construction) 5
1.2 Hierarchical structure 6
1.3 Virtual ring embedding on a rooted tree 8

4.1 Inconsistency range around a faulty process 42

5.1 Ring embedding on a tree 50
5.2 Preorder-postorder traversal 52
5.3 An example of causal shift 54
5.4 Faults in P_r 56
5.5 An example of $\mathcal{R}'(E_r)$ 58
5.6 Local routing function at p 59
5.7 Fault at p on the virtual ring 63
5.8 Faults at intermediate processes on the *virtual link* (q, p) 64
5.9 Majority values at process p 66
5.10 Fault at p in the virtual ring (Data a was delivered before the fault.) 67
5.11 Fault at p in the virtual ring (Data a was not delivered before the fault.) 71
5.12 The delay caused by RET 75
5.13 Embedding forward and backward rings 76
5.14 Leader election in the bidirectional ring 77

List of Tables

3.1 Notations for the source protocols and the composite protocol (*RWFC-LNS*) . . 22

4.1 Notations for the source protocols and the composite protocol (*RWFC-IcD*) . . . 41

Chapter 1

Introduction

1.1 Distributed Systems

A distributed system consists of computational entities that communicate with each other by communication links. (We call each computational entity *process* in the following.) Processes cooperate to accomplish the objectives of the system. Distributed systems model communication networks, multiprocessing computers, multitasking single computers, etc. There are fundamental expectations in designing distributed systems and distributed protocols, e.g. performance, scalability, availability, resource sharing, dependability, fault tolerance.

The difficulty in designing distributed systems lies in the distributed nature itself. The whole system should achieve its objectives, while each process has to compute with limited information about the entire system, e.g. the number of processes, topology of the network, independent input at each processes, asynchrony in the computation at each process or message delivery, and failures of many types.

Recently, the application of distributed systems has been growing rapidly, e.g. the Internet, peer-to-peer networks, mobile ad-hoc networks, sensor networks, inter-vehicle networks. These new applications introduce novel requirements on distributed systems. The Internet and peer-to-peer networks consist of a huge number of computers in the world. As the size of a network grows, dynamic changes (e.g. faults and topology changes) occur more frequently. The effect of a change may spread over the entire network due to the communication among processes. For example, a non-faulty process is affected by a fault by communicating with faulty processes and updating its state according to the information exchanged, and the effect may keep on spreading in the same way. This may damage the performance, availability, and dependability of the system. It is necessary for the system to automatically adapt to the changes and to prevent the effect of the changes from spreading over the entire network. Sensor networks

consist of a large number of sensor nodes and they are often operated under harsh environment. Thus, unexpected faults and topology changes can occur frequently. Because a large number of sensor nodes are distributed over wide field, it is desirable that the system adapts to the environment without human intervention. On the other hand, each sensor node has a small processor and small battery capacity. The difference between sensor networks and the Internet is this limitation on resource at each process. So, in sensor networks, it is desirable that the adaptability is achieved with small computational overhead. Inter-vehicle networks are exposed to dynamic topology changes and the biggest difference from above networks is mobile speed of each process. In such networks, it is expected that the system adapts to fast-changing topology quickly so that the system tolerates the next topology change.

1.2 Fault Tolerance of Distributed Systems

Distributed systems are prone to failures, e.g. memory contents at processes may be corrupted, processes may behave arbitrarily and may stop their actions, and messages exchanged between processes may be changed and lost. Fault tolerance is to mask the effect of failures or recover the objective behavior of a system after failures. As distributed systems play more critical role, fault tolerance of distributed systems is getting more and more important.

There are many levels of failures, e.g. hardware, software, process, communication. We focus on failures at processes and they are classified into the following four types [21, 41].

- **Crash failure.** A process stops its actions permanently when it undergoes a crash failure.
- **Omission failure.** In a network such that processes send and receive messages with each other, a receiver process does not receive some of the messages sent to it when it undergoes an omission failure.
- **Transient failure.** A transient failure changes the states of some processes arbitrarily by changing their memory contents arbitrarily.
- **Byzantine failure.** A Byzantine failure makes the process behave arbitrarily. This model is the strongest model of all process failure models.

Useful properties of distributed systems are classified into either *safety property* or *liveness property* [21, 41, 51]. Safety property implies that "bad things never happen" where bad things mean abnormal behavior of the system. Formally, safety property addresses that the system satisfies its safety specification in any execution. Liveness property implies that "good things eventually happen" where good things means the specification or purpose of the system. Formally, liveness property addresses that the system eventually satisfies its specification in any

execution. The term *eventuality* means a finite time, i.e. finite actions or computations. It is desirable that distributed systems always satisfy both safety property and liveness property. However, when failure occurs, these properties may be no longer satisfied. To design fault-tolerant distributed systems, it is necessary to guarantee that at least one of these properties is always satisfied even when failures occur.

There exist many approaches to promise fault tolerance. These approaches are classified into the following three types [21, 51].

- **Masking tolerance.** Masking tolerance promises that the application of the system does not observe the effect of failures and the system always satisfies its specification. Hence, the system always satisfies safety property and liveness property even when there exist failures.

- **Non-masking tolerance.** Non-masking tolerance allows that the application is temporally affected after failures, but eventually the effect ceases and the system behaves as its specification. Hence, the system always satisfies liveness property, however, when there exist failures, it does not promise safety property.

- **Fail-safe tolerance.** Fail-safe tolerance just avoids critical faulty configurations that damage the application, and even when failures occur, fail-safe tolerance may allow faulty configurations if it does not affect the application. Hence, the system always satisfies safety property, however, when there exist failures, it does not promise liveness property.

Masking fault tolerance is more preferable, however it is costly to implement masking fault-tolerant distributed systems. Non-masking fault tolerance provides a reasonable way of implementing fault-tolerant distributed systems.

1.3 Self-stabilization

To design fault-tolerant distributed systems, *self-*$*$ properties attract increasing attention, e.g. self-stabilizing, self-adaptive, self-configuring, self-healing, self-managing, self-organizing, self-optimizing, self-repairing. Self-* properties promise that the system automatically adapts to faults. Self-stabilization is one of the most promising design paradigms for adaptive fault-tolerant distributed protocols.

1.3.1 Self-stabilization and Adaptive Stabilization

Dijkstra [13] first introduced the notion of *self-stabilization* in 1974. Self-stabilization provides non-masking fault tolerance against finite number of transient faults that corrupts processes

by changing memory contents at processes arbitrarily. A self-stabilizing protocol promises that starting from any arbitrary initial configuration, the system eventually converges to a legitimate configuration where the protocol satisfies its specification. Hence, for a finite number of transient faults, self-stabilization promises autonomous adaptability by considering the configuration after the last fault as an initial configuration. A large number of self-stabilizing protocols have been proposed for many problems, e.g. spanning tree construction [11, 19, 30], leader election [22, 40], maximal independent set [50, 52], maximal matching [29, 42, 43], vertex coloring [28], median finding [9], synchronization [7, 32], propagation of information with feedback (PIF) [10, 12], token circulation [31], TDMA slot assignment [28], and routing [8]. Good surveys are found in [14, 20, 48, 50]. Self-stabilization is also widely used in real networks. IEEE 802.1d spanning tree protocol enables Ethernet bridges to construct a spanning tree in a self-stabilizing manner to avoid packet loops. RIP (Routing Information Protocol) based on Bellman-Ford routing algorithm and OSPF (Open Shortest Path First) based on periodical refresh of routing informations are also self-stabilizing.

Though self-stabilization achieves excellent fault tolerance against large scale faults, catastrophic faults rarely occur in practice while small scale faults are more likely to occur frequently. Moreover, self-stabilization guarantees nothing during the stabilization, and the effect of a small scale fault may spread over the entire network (Figure 1.1).

Many researchers have tried to develop adaptive self-stabilization by restricting the fault scenario, e.g. fault-containment [22, 23], time-adaptive stabilization [36], superstabilization [15, 34], local stabilization [1], and time-to-fault adaptive stabilization [16]. Fault-containment guarantees that when a fault corrupts at most f processes in a legitimate configuration, the system reaches a legitimate configuration in a time proportional in f. (The value of f depends on the protocol.) Time-adaptive stabilization guarantees that after a fault corrupts processes in a legitimate configuration, the system reaches a legitimate configuration in a time proportional to the number of corrupted processes. Super-stabilization guarantees that the system keeps its safety during the convergence after any topology change in a legitimate configuration. Local stabilization promises that after a fault corrupts processes in a legitimate configuration, the system reaches a legitimate configuration in a time proportional to the diameter of corrupted region. Time-to-fault adaptive stabilization guarantees that the output of the protocol recovers in a time proportional to the number of corrupted processes in an initial configuration. The main issue is the time complexity for recovery. Their aim is to guarantee the recovery time bounded by the number of corrupted processes in an initial configuration.

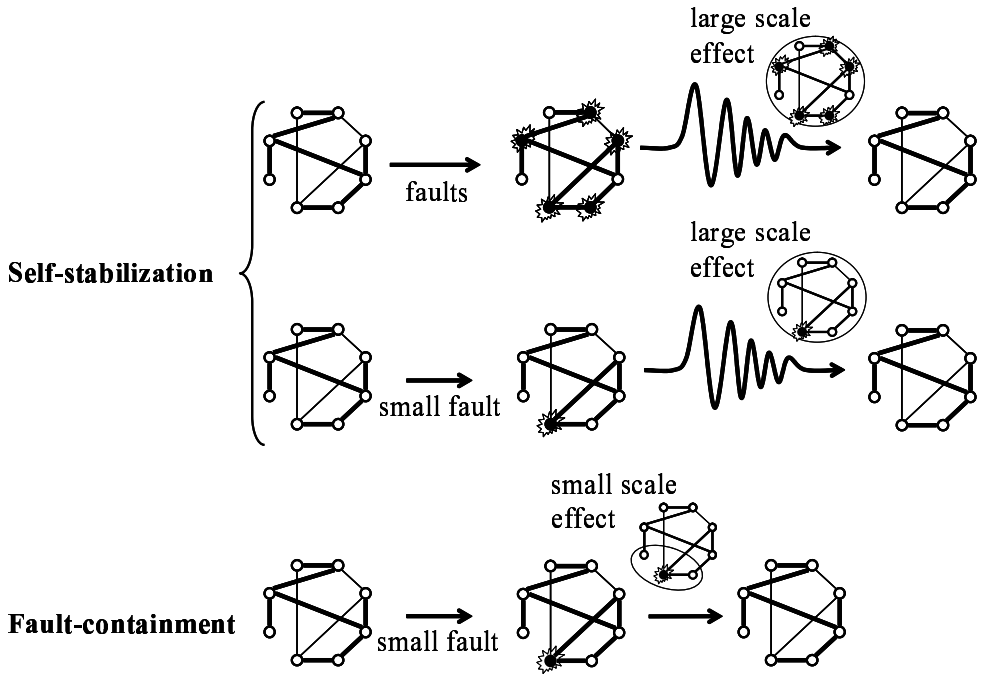


Figure 1.1: Self-stabilization and fault-containment (Spanning tree construction)

1.3.2 Fault-containment

Ghosh et al. [22, 23] first introduced the notion of *fault-containment* in 1996. An f-faulty configuration is a configuration obtained by a fault corrupting f processes in a legitimate configuration. An f-fault-containing protocol promises self-stabilization against large scale faults and containment of the effect against small scale faults, i.e. for any f'-faulty configuration, where f' is smaller than or equals to f, the effect is contained in any execution starting from the configuration. The containment property is twofold: one is *spatial containment* property that promises that the effect of the fault is contained around faulty processes. The other is *temporal containment* property that promises that the effect of the fault lasts just a short period of time after the fault (Figure 1.1). There already exist many fault-containing protocols, e.g. a small scale fault is contained and rapid recovery is guaranteed in the fault-containing protocols for rings [22, 26], and for general graphs [23, 24, 25, 39]. Ghosh et al. proposed fault-containing leader election on rings [22], k-fault-containing token circulation [26], 1-fault-containing BFS tree construction [24], 1-fault-containing spanning tree construction [25], and 1-fault-containing maximal independent sets [39]. Some of the above fault-containing protocols are obtained by adding fault-containment property to already existing self-stabilizing protocols. Ghosh et al. present a general technique for adding 1-fault-containment property to non-reactive

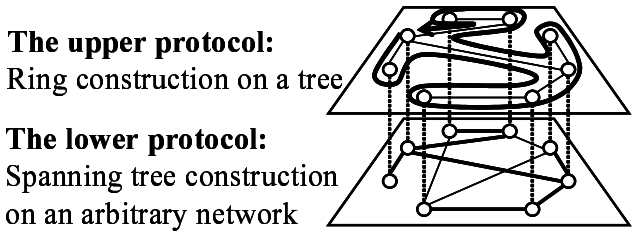


Figure 1.2: Hierarchical structure

self-stabilizing protocols [23]. Ghosh et al. introduced 1-fault-containing spanning tree construction using priority scheduler in [25]. Priority scheduler provides a weak priority rule that makes the recovery actions of faulty processes precede the actions of correct processes. There exist such fault-containing protocols obtained by composing multiple layers of protocols where each protocol is not fault-containing by itself [4, 3]. However, these transformers are designed for limited fault scenarios.

1.4 Hierarchical Design of Distributed Protocols

Hierarchical design of protocols improves the reusability of existing protocols by extending the application of these protocols and eases the design of new protocols. Hierarchy means that the output of one protocol (the lower protocol) is used as the input to the other protocol (the upper protocol). For example, Figure 1.2 shows a hierarchical structure of two protocols where the lower protocol is a spanning tree construction on an arbitrary graph and the upper protocol is a ring embedding on an arbitrary tree. The tree constructed by the lower protocol is used as the input by the upper protocol. In this dissertation, we focus on two types of hierarchical design of self-stabilizing protocols. First, we introduce hierarchical composition of self-stabilizing protocols that facilitates the design of new protocols. Secondly, we introduce topology embedding that extends the application of existing protocols designed for a specific topology to another topology.

1.4.1 Composition

In a hierarchical composition of two (or more) distributed protocols, the output of one protocol (the lower protocol) is used as the input to the other protocol (the upper protocol) and the composite protocol provides the output of the upper protocol on the input to the lower protocol. We call the lower protocol and the upper protocol source protocols. Hierarchical composition eases the design of new protocols and improves the reusability of existing protocols.

Hierarchical composition of fault-tolerant distributed protocols has been well studied. Gouda et al. proposed an *adaptive programming* for the systems with input changes [27]. They proposed *hierarchical composition* of adaptive protocols that forces each process to execute the lower protocol first so that the process executes the upper protocol after the lower protocol. Their hierarchical composition checks whether a process has to execute the lower protocol and only when it does not have to execute the lower protocol, the process can execute the upper protocol. However, in general, this local checking of the lower protocol cannot guarantee completion of the global recovery of the lower protocol when the upper protocol starts its execution.

Hierarchical composition of self-stabilizing protocols is also well used in the design of new self-stabilizing protocols. One of the most well-known composition techniques for self-stabilizing protocols is *fair-composition* [17, 19]. Fair-composition executes source protocols in parallel and it guarantees that the composite protocol is also self-stabilizing. Starting from an arbitrary initial configuration, the lower protocol first reaches a legitimate configuration with its self-stabilizing property. Though the upper protocol can be also executed on the incorrect input from the lower protocol during the convergence of the lower protocol, self-stabilization guarantees convergence from any initial configuration. Hence, after the lower protocol reaches a legitimate configuration, the upper protocol eventually reaches a legitimate configuration and the composite protocol eventually reaches a legitimate configuration.

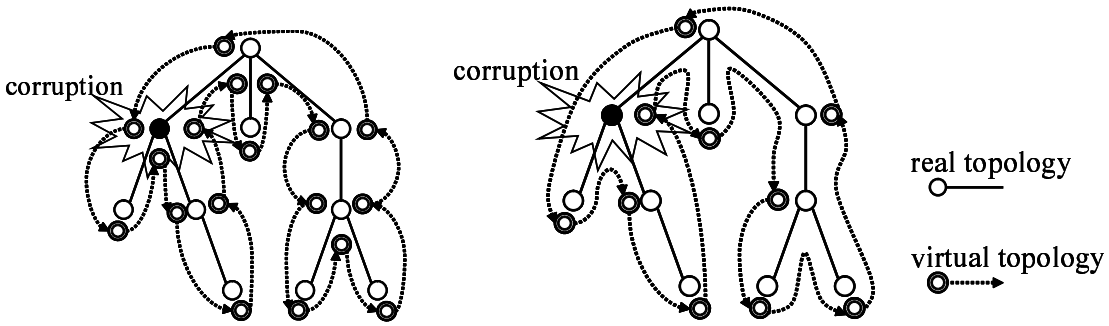
Other than hierarchical structures, many composition techniques for self-stabilizing protocols have been also developed. Gouda et al. [27] also proposed *selective composition* that executes multiple adaptive protocols and switches the output so that the composite protocol can adapt to input changes. Beauquier et al. [6] introduced *cross-over composition* which uses the lower protocol as a filter to the execution of the upper protocol and improves the adaptability to scheduler. Dolev et al. [16] proposed *parallel composition* that enables parallel search and accelerates the stabilization by executing multiple self-stabilizing protocols in parallel.

1.4.2 Topology Embedding

Topology embedding is to embed a virtual topology on a real topology that enables a distributed protocol designed for a specific topology (virtual topology) to be executed on another topology (real topology) and extends the application of the protocol.

We can find two types of topology embedding [1]. *Many-to-one node embedding* is an em-

[1] We can find *one-to-many node embedding* in [47]. In [47], Nolte et al. proposed *virtual node layer* for mobile ad-hoc networks that deploys virtual nodes on the predefined geographic coordinates. Virtual nodes are realized by the mobile nodes around the predefined points. Thus, virtual node layer enables stable deployment of virtual nodes in mobile ad-hoc networks even when mobile nodes move.



(a) Many-to-one node embedding (b) One-to-one node embedding

Figure 1.3: Virtual ring embedding on a rooted tree

bedding such that one real process corresponds to multiple virtual processes. *One-to-one node embedding* is an embedding such that one real process corresponds to just one virtual process and different real processes correspond to different virtual processes. Figure 1.3 shows an example of ring embedding on a rooted tree. Based on the depth-first traversal, Figure 1.3 (a) shows a many-to-one node embedding while Figure 1.3 (b) shows a one-to-one node embedding. Consider a fault that corrupts one real process in Figure 1.3. In many-to-one node embedding, corresponding multiple virtual processes are corrupted in the virtual topology. On the other hand, in one-to-one node embedding, just one corresponding virtual process is corrupted in the virtual topology. In this way, any one-to-one node embedding has natural fault tolerance because when a fault corrupts f real processes, it corresponds to a situation where f virtual processes are corrupted by the fault in the virtual topology. To preserve the fault-containment property in the virtual topology, this fact is very useful because the containment guarantee depends on the number of processes corrupted by a fault.

However, in a one-to-one node embedding, each virtual link between two virtual processes may be a path between corresponding real processes in the real topology. *Dilation* is the maximum distance of a virtual link in the real topology. A one-to-one node embedding can introduce dilation larger than one. For example, Sekanina [49] proposed one-to-one node embedding for ring on an arbitrary tree that has the dilation of three. Hence, each data read through (or sent and received on) a virtual link should be relayed by the intermediate processes in a real topology. When a real process is corrupted by a fault, the virtual links running through the corrupted process can be also corrupted. However, in general, fault tolerance against unreliable communication links is easier than fault tolerance against corruption at processes. For example, by duplicating messages or attaching sequence numbers to messages, we can avoid message loss or message duplication.

1.5 Overview of This Dissertation

In this dissertation, we focus on hierarchical design of fault-containing protocols that facilitates the design of new fault-containing protocols and extends the application of existing fault-containing protocols. We propose two methods to realize hierarchical structures of fault-containing protocols.

1.5.1 Hierarchical Composition of Fault-containing Protocols

When designing hierarchical composition of fault-containing protocols, the main concern is to preserve the fault-containment property of source protocols. Though several composition methods for self-stabilizing protocols have been proposed [17, 19, 27], existing composition methods do not preserve the fault-containment property of source protocols. Fair composition [17, 19] of self-stabilizing protocols cannot preserve the fault-containment property of source protocols. This is because the parallel execution of the source protocols allows the upper protocol to be executed on an incorrect output of the lower protocol. Then, the effect of a fault may spread over the entire network in the upper protocol. Hierarchical composition [27] of self-stabilizing protocols also cannot preserve the fault-containment property of source protocols. Hierarchical composition just checks whether a process has to execute the lower protocol (i.e. whether it has an enabled guard in the lower protocol) and only when it does not have to execute the lower protocol, the process can execute the upper protocol. The problem is that we cannot guarantee the overall recovery of the lower protocol by checking whether one process has an enabled guard in the lower protocol or not. The difficulty in composing fault-containing protocols lies in how to guarantee the recovery of the lower protocol when processes execute the upper protocol.

For any f_1-fault-containing protocol P_1 and f_2-fault-containing protocol P_2, a hierarchical composition of P_1 and P_2 is a *fault-containing composition* when the composite protocol is $f_{1,2}$-fault-containing for some $0 < f_{1,2} \leq f_1, f_2$. We propose *Recovery Waiting Fault-containing Composition* (*RWFC*) strategy that stops the upper protocol during the recovery of the lower protocol. To implement *RWFC* strategy, we utilize the containment properties of fault-containing protocols. Temporal containment property provides the *recovery time* that is the maximum time necessary for the system to recover from any target faulty configuration. The first composition technique *RWFC-LNS* (*RWFC* with the local neighborhood synchronizer) stops the upper protocol for the recovery time of the lower protocol. After its recovery time, the lower protocol is in a legitimate configuration, and the upper protocol recovers with its fault-containment property. We also implement the *local neighborhood synchronizer* (*LNS*) that measures time in asynchronous distributed systems while preserving the spatial and temporal containment (i.e.

LNS is executed only around faulty processes and only a short period of time after the fault). However, in the worst case, *RWFC-LNS* stops the upper protocol even when the lower protocol has recovered. Moreover, not all fault-containing protocols provide both temporal containment property and spatial containment property. They just provide temporal containment property and/or spatial containment property. Spatial containment property provides the *inconsistency range* that is the maximum distance from any faulty process to any process that finds inconsistency during the recovery of the protocol. The second composition technique *RWFC-IcD* (*RWFC* with inconsistency detector) utilizes the inconsistency range to detect the recovery of the lower protocol. *RWFC-IcD* enables the upper protocol to start its recovery as soon as the lower protocol recovers and speeds up the recovery of the composite protocol. We implemented the *inconsistency detector* (*IcD*) that detects the inconsistency of the lower protocol.

The proposed composition techniques are important both theoretically and practically. These composition techniques suggest the possibility of a uniform framework for composition of fault-containing protocols and a novel design technique for fault-containing protocols.

1.5.2 Ring Embedding

As one of the most investigated networks in distributed computing, a ring network is frequently used for distributed computation and control. Dijkstra designed the first self-stabilizing protocols for ring networks (three mutual exclusion protocols in [13]). The election problem, one of the most fundamental problems, was first introduced by Le Lann for ring networks in a non-self-stabilizing manner [38]. Fault-containing ring protocols are also proposed (leader election [22], and token circulation [26]). A substantial advantage of ring protocols is that they can be applied to arbitrary networks by means of virtual rings embedded on the real networks.

Kulkarni et al. proposed a transformation technique using a virtual ring embedded on a spanning tree [35]. Their transformation enables self-stabilizing protocols designed for theoretical models to be executed on a *write all with collision* (WAC) model. WAC model reflects collisions of broadcasts in sensor networks and a virtual ring is used for mutual exclusion to avoid collisions. However, their ring embedding is a many-to-one node embedding.

One effective way of designing fault-containing protocols is to apply existing ones, designed for simple networks (e.g. rings), to arbitrary networks. This approach is common in protocol design. However, to the best of our knowledge, this approach has not been investigated in the context of fault-containment.

We propose a one-to-one ring embedding on an arbitrary rooted tree that preserves the fault-containment property of ring protocols executed on the embedded ring. To tolerate the corruption of virtual links, we implement the communication mechanism that enables the cor-

rupted data to be discarded at endpoint virtual processes.

Simulation is to provide the same task as an original protocol designed for a specific computation model on another computation model. Lynch defined the *simulation relation* between two different protocols that requires one protocol traces every global configuration of the other protocol [41]. In our ring embedding, it is difficult to simulate the global configurations of the original ring protocol because virtual links have different communication delays. However, our method preserves the read/write causality of the original ring protocol that makes us call our method *causal simulation*. Though causal simulation is a weaker notion than simulation, it is strong enough to guarantee that the simulating protocol can execute the same task as the original protocol. Causal simulation provides simulation of ring protocols for non-reactive tasks (e.g. leader election, etc.) and reactive tasks (e.g. token circulation, etc.) such that the safety property of the task depends only on the read/write causality. Since most of the reactive protocols are based on read/write causality, causal simulation can be applied to a variety of protocols.

To the best of our knowledge, this ring embedding method is the first challenge to develop a method to simulate a protocol on another topology while preserving the fault-containment property. Using the fault-containing composition and existing fault-containing spanning tree construction [24, 25], the proposed method can be extended to arbitrary networks. Consequently, this work pioneers a new methodology of designing fault-containing protocols on arbitrary networks.

Though our framework focuses on the ring embedding on rooted trees, this embedding technique suggests the possibility of uniform topology embedding technique for simulating fault-containing protocols.

1.6 Organization of This Dissertation

This dissertation consists of six chapters. In Chapter 2, we give formal definitions of computational models and self-stabilization. In Chapter 3 and 4, we introduce the notion of fault-containing composition and show two different hierarchical composition techniques that preserves the fault-containment property of source protocols. In Chapter 5, we show ring embedding on an arbitrary rooted tree that preserves the fault-containment property of ring protocols executed on an embedded ring. We conclude this dissertation in Chapter 6.

Chapter 2

Preliminary

2.1 Network and Processes

A system is a network which is represented by a undirected graph $G = (V, E)$ where the vertex set V is a set of processes and the edge set E is a set of bidirectional communication links. Each process has a unique identity. Process p is a neighbor of process q iff there exists a bidirectional communication link $(p, q) \in E$. A set of direct neighbors of p is denoted by N_p and $\delta_p = |N_p|$ is the degree of p. Let $N_p^1 = N_p$, and for each $i \geq 2$, $N_p^i = N_p^{i-1} \cup \bigcup_{q \in N_p^{i-1}} N_q \backslash \{p\}$. The set of processes denoted by N_p^i is called *i-neighbor* of p. The i-neighbor of p is the set of processes such that their distances from p are smaller than or equal to i excluding p. The distance between p and q ($q \neq p$) is denoted by $dist(p, q)$, and $dist(p, q) = j$ iff $q \notin N_p^{j-1} \wedge q \in N_p^j$.

Each process p maintains local variables and the values of all local variables at p define the local state of p. Local variables are classified into three classes: input, output, and inner. The input variables indicate the input to the system and they are not changed by the system. The output variables are the output of the system for external observers. The inner variables are internal working variables used to compute output variables.

We adopt *locally shared memory model* [1] as a communication model: each process p can read the values of the local variables at $q \in N_p \cup \{p\}$. A protocol at each process p consists of a finite number of *guarded actions* in the form of $\langle guard \rangle \rightarrow \langle action \rangle$. A $\langle guard \rangle$ is a boolean expression involving the local variables of p and N_p, and an $\langle action \rangle$ is a statement that changes the values of p's local variables (except input variables). A process with a guard evaluated to *true* is called

[1] There exist *message passing model* and *link register model*. In message passing model, processes communicate with each other by sending and receiving messages. Link register model models each link as a register and each message is written to and read from the register. However, the idea presented in this dissertation does not depend on the communication model. Many researchers have tried to transform a protocol designed for the locally shared memory model into a protocol on other communication models [19, 18, 53].

enabled. We adopt *distributed daemon* as a scheduler: in a computation step, distributed daemon selects a nonempty subset of enabled processes, and each selected process executes one of the corresponding actions. We consider the distributed daemon is weakly fair, that is, if a process evaluates some of its guards to be *true* infinitely often, the process is selected by the distributed daemon infinitely often. The evaluation of guards and the execution of the corresponding action is *atomic*: these computations are done without any interruption. A configuration of a system is represented by a tuple of local states of all processes. An *execution* is a maximal sequence of configurations $E = \sigma_0, \sigma_1, \sigma_2, \cdots$ that satisfies (i) σ_{i+1} is obtained by applying one computation step to σ_i or (ii) σ_i is the final configuration. Maximality means that the sequence is either infinite, or it is finite and no process is enabled in the final configuration.

Distributed daemon allows *asynchronous* executions. In an asynchronous execution, the time is measured by computation steps or *rounds*. Let $E = \sigma_0, \sigma_1, \sigma_2, \cdots$ be an asynchronous execution. The first round $\sigma_0, \sigma_1, \sigma_2, \cdots, \sigma_i$ is the minimum prefix of E such that for each process $p \in V$ if p is enabled in σ_0, either p's guard becomes disabled or p executes at least one step in $\sigma_0, \sigma_1, \sigma_2, \cdots \sigma_i$. The second and latter rounds are defined recursively by applying the definition of the first round to the remaining suffix of the execution $E' = \sigma_{i+1}, \sigma_{i+2}, \cdots$.

2.2 Self-stabilization

A *problem* (task) $\mathcal{T}$ is defined by a legitimate predicate on configurations. A *non-reactive* problem is a problem such that no process changes the values of its output variables after the system reaches a configuration where the legitimate predicate holds, e.g. spanning tree construction, and leader election. A *reactive* problem is a problem such that processes change the values of their output variables after the system reaches a configuration where the legitimate predicate holds, e.g. token circulation and synchronization.

There exist two types of definitions for legitimacy. The difference is with what the legitimacy is defined: one defines legitimacy based on configurations and the other defines legitimacy based on executions. It is hard to define reactive problems by legitimacy defined on configurations. For example, it is difficult to determine liveness and fairness of the token circulation problem with one configuration. For the token circulation problem, the legitimacy should be defined with executions.

Definition 1 ***Legitimate Configuration of Non-reactive Problems***

For a non-reactive problem $\mathcal{T}$, a configuration σ is legitimate *iff σ satisfies the legitimate predicate of $\mathcal{T}$.*

Definition 2 ***Legitimate Configuration of Reactive (and Non-reactive) Problems***
For a reactive (and non-reactive) problem $\mathcal{T}'$, a configuration σ is legitimate *iff any configuration that appears in any execution starting from σ satisfies the legitimate predicate of $\mathcal{T}'$.*

We say a distributed protocol $P(\mathcal{T})$ has solved problem $\mathcal{T}$ in a configuration iff the configuration satisfies the legitimate predicate $L(P(\mathcal{T}))$. The input (output) of $P(\mathcal{T})$ is represented by the conjunction of input (output, respectively) variables at each process. We omit $\mathcal{T}$ if $\mathcal{T}$ is clear. The input variables to the protocol are not changed during the execution of the protocol.

Definition 3 ***Self-stabilization***
A distributed protocol P is self-stabilizing iff it satisfies the following two properties:

- ***Convergence*** *: starting from any arbitrary initial configuration, it reaches a legitimate configuration.*
- ***Closure*** *: once it reaches a legitimate configuration, it remains in legitimate configurations thereafter.*

A transient fault corrupts some processes by changing the values of their local variables (except input variables) arbitrarily. A self-stabilizing protocol guarantees autonomous adaptability against any finite number of transient faults by considering the configuration after the last fault as an arbitrary initial configuration from that it starts the convergence.

A configuration is *f-faulty* [2] iff the minimum number of processes such that we have to change their local states (except input variables) to make the configuration legitimate is f. So, an f-faulty configuration is the configuration just after a fault corrupts f processes. We say process p is *faulty* iff we have to change p's local state to make the configuration legitimate and otherwise *correct*.

An *f-fault-containing* protocol autonomously reaches a legitimate configuration from any f'-faulty configuration ($f' \leq f$) in a polynomial time in f, and/or the number of processes affected is bounded by a polynomial in f, e.g. f, f^2 (not $|V|$). We say a processes is *contaminated* iff the process changes its local variables during the recovery from an f'-faulty configuration ($f' \leq f$).

Definition 4 ***f-fault-containment***
A self-stabilizing protocol is f-fault-containing iff it reaches a legitimate configuration from any f'-faulty configuration ($f' \leq f$) with the number of contaminated processes and/or the number of rounds to reach a legitimate configuration bounded by some polynomial in f (not $|V|$).

[2] In general, the legitimate configuration obtained by changing the local states of f processes is not always unique. However, in this dissertation we assume the corresponding legitimate configuration is unique because we assume later that the legitimate configuration of the protocol is uniquely defined by the input and the input is not corrupted by faults.

We simply denote an f-fault-containing self-stabilizing protocol as f-fault containing protocol.

Fault-containing protocols promise two types of containment, *spatial containment* and/or *temporal containment*. The performance of an f-fault-containing protocol is measured by the following criteria.

Stabilization :

- **Stabilization time** : the maximum (worst) number of rounds to reach a legitimate configuration from an arbitrary initial configuration.

Spatial containment :

- **Contamination radius** : the maximum distance from any faulty process to the process that changes its local state according to the faulty process during the recovery from an f'-faulty configuration ($f' \leq f$).
- **Contamination number** : the maximum (worst) number of contaminated processes from an f'-faulty configuration ($f' \leq f$).

Temporal containment :

- **Recovery time** : the maximum (worst) number of rounds to reach a legitimate configuration from an f'-faulty configuration ($f' \leq f$).

These performance criteria bound the effect after faults: starting from an arbitrary initial configuration, the protocol has reached a legitimate configuration after its stabilization time. Starting from an f'- faulty configuration ($f' \leq f$), the protocol has reached a legitimate configuration after its recovery time. Also, during the recovery from an f'-faulty configuration, the number of contaminated process is at most its contamination number or the distance from any contaminated process to a faulty process that caused the contamination is at most its contamination radius.

Chapter 3

Hierarchical Composition with Temporal Containment

In this chapter, we present a timer-based hierarchical composition technique for fault-containing protocols that preserves fault-containment property of source protocols. In a hierarchical composition of two (or more) distributed protocols, the output of one protocol (the lower protocol) is used as the input to the other protocol (the upper protocol) and the composite protocol provides the output of the upper protocol on the input to the lower protocol. We call the lower protocol and the upper protocol source protocols. A composition is called *fault-containing composition* if it preserves the fault-containment property of the source protocols.

The proposed strategy is to control the execution of source protocols. What we call *RWFC* strategy (Recovery Waiting Fault-containing Composition) is to stop the upper protocol until the lower protocol recovers so that the upper protocol recovers with a correct input from the lower protocol. The difficulty lies in how to detect the recovery of the lower protocol.

The proposed composition technique utilizes temporal containment property of fault-containing protocols to control the execution of source protocols. We call the proposed composition technique *RWFC-LNS* (*RWFC* with the Local Neighborhood Synchronizer). The *recovery time* of a fault-containing protocol is the maximum time for the system to recover from a target faulty configuration. We force the upper protocol to stop during the recovery time of the lower protocol and the upper protocol always executes on the correct input from the lower protocol. Thus, it is guaranteed that the upper protocol recovers with its fault-containment property because it is suspended until the lower protocol recovers, and the composite protocol promises fault-containment as a whole.

To implement *RWFC-LNS*, it is necessary to measure time in an asynchronous system in a fault-containing manner. Our framework uses local timers at processes to measure the recovery

times of the source protocols. *Global synchronizers* are often used to implement timers that involve all processes into the synchronization. Ghosh et al. proposed a transformer for self-stabilizing protocols to obtain corresponding 1-fault-containing protocols [23]. An obtained 1-fault-containing protocol guarantees that the output of the protocol recovers quickly. However, their transformer utilizes a global neighborhood synchronizer and the effect of a fault spreads over the entire network via global synchronization. Though their transformer guarantees temporal containment and spatial containment only for the output of the obtained protocol, the protocol should wait the global synchronization to finish so that it can tolerate the next fault. The global neighborhood synchronizer cannot promise the spatial containment of the composite protocol because it involves all processes into the synchronization. To preserve the fault-containment property, we introduce a *local neighborhood synchronizer* that synchronizes a limited number of processes during a short period of time after a fault without involving all processes into the synchronization.

Related Works. One of the most commonly used hierarchical composition technique for self-stabilizing protocols is *fair composition*. Fair composition executes two (or more) different self-stabilizing protocols in parallel and promises self-stabilization of the composite protocol [17]. However, if we compose fault-containing protocols by fair composition, the composite protocol cannot preserve the fault-containment property of source protocols. This is because the parallel execution of the source protocols allows the upper protocol to be executed on an incorrect output of the lower protocol. Consider a fair composition of f_1-fault-containing protocol P_1 and f_2-fault-containing protocol P_2. When a fault corrupts the output variables of the lower protocol P_1 at f processes ($f \leq \min\{f_1, f_2\}$), during the recovery of P_1, the upper protocol P_2 can be executed in parallel to adopt the changes in the output variables of P_1. During the recovery of P_1, processes around each faulty process may change their states (possibly) repeatedly in P_1. If they change the value of their output variables of P_1, the input to P_2 also changes. If the number of such processes is greater than f_2, P_2 cannot guarantee fault-containment. Even when the number of such processes is smaller than f_2, if these processes change their outputs of P_1 repeatedly, P_2 cannot promise fault-containment. This is because a fault-containing protocol assumes that the input does not change during the recovery.

Gouda et al. proposed *adaptive programming* for the systems with input changes [27]. They proposed *hierarchical composition* of adaptive protocols that forces the lower protocol to be executed first so that it provides the stable input to the upper protocol. Their hierarchical composition just checks whether a process has to execute the lower protocol (i.e. whether it has an enabled guard in the lower protocol) and only when it does not have to execute the lower protocol, the process can execute the upper protocol. Though self-stabilization is one subclass

of adaptive protocols, hierarchical composition of self-stabilizing protocols cannot preserve the fault-containment property of source protocols. The problem is that we cannot guarantee the overall recovery of the lower protocol by locally checking whether one process has an enabled guard in the lower protocols or not.

These existing composition techniques cannot preserve the fault-containment property of source protocols because they do not guarantee the recovery of the lower protocol when the upper protocol is executed, and the effect of a fault can spread over the entire network in the upper protocol.

This chapter is organized as follows. In Section 3.1, we first give the formal definition of the fault-containing composition and introduce *RWFC* strategy. In Section 3.2, we show assumptions on the source protocols for *RWFC-LNS*. In Section 3.3, we first define the specification of the local neighborhood synchronizer and then present the composition framework *RWFC-LNS*. The correctness proof of *RWFC-LNS* is also shown in Section 3.3. In Section 3.4, we present an implementation of the local neighborhood synchronizer, protocol *LNS* and prove that *LNS* satisfies the specification in Section 3.3. We conclude this chapter with Section 3.5.

3.1 Fault-containing Composition and RWFC Strategy

We consider self-stabilization and fault-containment of protocols for non-reactive problems. Hence, the set of legitimate configurations of a problem is defined by Definition 1.

A hierarchical composition of two protocols P_1 and P_2 is denoted by $(P_1 * P_2)$ where the variables of P_1 and those of P_2 are disjoint except that the input to P_2 is the output of P_1. We define the output variables of $(P_1 * P_2)$ is the output variables of P_2. A legitimate configuration of $(P_1 * P_2)$ is defined by $L((P_1 * P_2))$ where $L(P_1 * P_2) = L(P_1) \wedge L(P_2)$.

Definition 5 ***Fault-containing composition***
Let P_1 be an f_1-fault-containing protocol and P_2 be an f_2-fault-containing protocol. A hierarchical composition $(P_1 * P_2)$ is a fault-containing composition of P_1 and P_2 iff $(P_1 * P_2)$ is an $f_{1,2}$-fault-containing protocol for some $f_{1,2}$ such that $0 < f_{1,2} \leq \min\{f_1, f_2\}$.

We call P_1 and P_2 the source protocols. Fault-containing composition preserves the fault-containment property of source protocols because $0 < f_{1,2} \leq \min\{f_1, f_2\}$ holds for an f_1-fault-containing protocol P_1 and an f_2-fault-containing protocol P_2.

In a hierarchical composition, the input to P_2 can be corrupted by a fault when the fault corrupts the output variables of P_1.

Remark 1 *For a hierarchical composition $(P_1 * P_2)$, the input to P_1 is not corrupted by any fault.*

The input to P_1 is given as the input variables at each process and we defined the input variables are not changed by any fault. Generally, fault-containment for non-reactive problems is designed under the assumption that the input to the protocol is not changed by the fault. The input to P_1 is considered as the system parameters, e.g. topology, ID of each process, etc.

For fault-containing composition, we consider a subclass of fault-containing protocols Π such that each f-fault-containing protocol $P \in \Pi$ satisfies Assumption 1. Many existing fault-containing protocols [22, 25] satisfy this assumption.

Assumption 1 *The legitimate configuration of P is uniquely defined by the input variables.*

Consider a composition $(P_1 * P_2)$ of an f_1-fault-containing protocol P_1 and an f_2-fault-containing protocol P_2. Starting from an f'-faulty configuration ($f' \leq \min\{f_1, f_2\}$), if the output of P_1 after P_1 reaches a legitimate configuration is different from what it was before the fault, then the input to P_2 is considered to change and it may appear to be an f'-faulty configuration for some $f' > min\{f_1, f_2\}$. Then, P_2 cannot guarantee fault-containment even though the original fault is small enough for the fault-containment of P_2. Because the input to P_1 is not changed by any fault (Remark 1), Assumption 1 guarantees that P_1 recovers to the unique legitimate configuration and ensures the possibility of fault-containment in the composite protocol.

Our approach to fault-containing composition is to control the execution of P_1 and P_2 to guarantee the recovery of P_1 when P_2 starts its execution. We call this approach *Recovery Waiting Fault-containing Composition* (*RWFC*).

Definition 6 ***RWFC strategy***
RWFC strategy is to stop the execution of P_2 until P_1 provides a correct output when $(P_1 * P_2)$ starts from a target faulty configuration.

To preserve the fault-containment (i.e. spatial containment and/or temporal containment) of source protocols, the implementation of *RWFC* strategy should have the following property.

Remark 2 *A composing protocol that realizes fault-containing composition should be also fault-containing.*

RWFC strategy preserves the fault-containment of P_1 and P_2 in the following way: starting from an f-faulty configuration ($f \leq f_{1,2}$), when P_1 reaches its unique legitimate configuration for the stable input [1], there are at most f faulty processes in P_2. Then P_2 can recover with its

[1] A fault cannot change the input variables of P_1 (Assumption 1) and P_1 reaches the unique legitimate configuration (Assumption 1).

fault-containment property and the whole composite protocol succeeds in containing the effect of faults.

3.2 Preliminary

In this chapter, we consider self-stabilization and fault-containment of protocols for non-reactive problems. Hence, the set of legitimate configurations of a problem is defined by Definition 1.

In this chapter, we consider a subclass of fault-containing protocols Π such that each f-fault-containing protocol $P \in \Pi$ satisfies Assumption 2 and 3. Many existing fault-containing protocols [22, 25] satisfy these assumptions.

Assumption 2 *The legitimate predicate $L(P)$ for P is represented in the form $L(P) \equiv \forall p \in V : cons_p(P)$. The predicate $cons_p(P)$ involves the local variables at p and its neighbors, and it is defined over the values of output, inner, and input variables.*

We say process p is *inconsistent* iff $cons_p(P)$ is evaluated to *false* at p, otherwise *consistent*. Because we work on non-reactive problems, the predicate $cons_p(P)$ is evaluated to *false* when process p is enabled.

Assumption 3 *In an f'-faulty configuration ($f' \leq f$), if a faulty process p is a neighbor of correct process(es), at least one correct process q neighboring to p evaluates $cons_q(P)$ to false or p evaluates $cons_p(P)$ to false.*

For a faulty process p and a neighboring correct process q, $cons_p(P)$ ($cons_q(P)$, respectively) involves the local variables at q and p. Because p is faulty, there can be some inconsistency between the local state of p and that of q.

Assumption 4 *The recovery time of f-fault-containing protocol P is larger than its contamination radius and f.*

Generally, the recovery time of an f-fault-containing protocol is not always larger than f and the contamination radius in an asynchronous system. However, in synchronous systems, the recovery time is always larger than f and the contamination radius. Because our composition technique executes the source protocols in a synchronous manner, we put this assumption.

3.3 Composition Framework

Let P_1 be an f_1-fault-containing protocol and P_2 be an f_2-fault-containing protocol. Our goal is to produce $f_{1,2}$-fault-containing protocol $(P_1 * P_2)$ for $f_{1,2} = \min\{f_1, f_2\}$.

Table 3.1: Notations for the source protocols and the composite protocol (*RWFC-LNS*)

protocol	number of maximum faults	recovery time	contamination radius
P_1	f_1	r_1	c_1
P_2	f_2	r_2	c_2
$(P_1 * P_2)$	$f_{1,2} = \min\{f_1, f_2\}$	$r_{1,2}$	$c_{1,2}$

In this chapter, we use the notations shown in Table 3.1.

RWFC strategy is a strategy for fault-containing composition: P_2 should wait the recovery of P_1. The key is how to guarantee the recovery of P_1. To implement *RWFC* strategy, the proposed fault-containing composition utilizes the recovery time of fault-containing protocols. Starting from an f-faulty configuration ($f \leq f_{1,2}$), if a process finds inconsistency in P_1 or P_2, the process stops the execution of P_2 at processes in the contamination radius of P_1 and P_2 for r_1 rounds. (Note that the inconsistency in P_2 may be caused by the corruption of output variables of P_1.) During the r_1 rounds, these processes execute only P_1 and P_1 reaches a legitimate configuration. After that, these processes execute P_2 on the correct input from P_1.

To make the composite protocol fault-containing, it is necessary that all processes in the contamination radius from each faulty process measure time from f-faulty configuration, i.e. these processes need *local timers*. We implement local timers at processes with a *local neighborhood synchronizer* that synchronizes the processes in $\max\{c_1, c_2\}$-neighbors for each faulty process for $(r_1 + r_2)$ rounds.

The idea of our composition is as follows: from Assumption 3, in an f-faulty configuration ($f \leq f_{1,2}$), if faulty process p has a correct process q in its neighbor, p or q finds the inconsistency between them in P_1 or P_2. The proposed composition technique utilizes this property. So, when process s finds inconsistency with its neighbor(s) in P_1 or P_2, it triggers the synchronization of the local neighborhood synchronizer. The maximum distance from s to any contaminated process is $\max\{c_1, c_2\} + \min\{f_1, f_2\} + 1$. (Note that the target faulty configuration is f-faulty configuration for $f \leq f_{1,2} = \min\{f_1, f_2\}$.) Then, all processes in $N_r^{\max\{c_1,c_2\}+\min\{f_1,f_2\}+1}$ are involved in the synchronization and these processes execute P_1 for the first r_1 rounds and P_2 for the next r_2 rounds.

For the fault-containment of the composite protocol, it is necessary that no correct process executes P_2 before P_1 recovers. If a correct process executes P_2 before P_1 recovers, the number of faulty processes in P_2 may become larger than f_2. On the other hand, we can allow faulty processes to execute P_2 before P_1 reaches a legitimate configuration because in an f-faulty

configuration ($f \leq f_{1,2}$), even if faulty processes execute P_2 before P_1 recovers, the number of faulty process in P_2 is still no larger than f.

We first define the specification of the local neighborhood synchronizer in Section 3.3.1 and show our composition framework in Section 3.3.2. The proof for the framework with the local neighborhood synchronizer is shown in Section 3.3.3.

3.3.1 Specification of the Local Neighborhood Synchronizer

In this section, we define the specification of the local neighborhood synchronizer for fault-containing composition ($P_1 * P_2$).

Specification 1 ***Stabilization***
Each process p maintains a timer variable t_p that takes an integer in $[0..(r_1 + r_2)]$. The local neighborhood synchronizer is self-stabilizing and in a legitimate configuration, $t_p = 0$ holds at each $p \in V$.

The local neighborhood synchronizer is implemented with a typical technique of synchronizers [23]. We say a process is *s-consistent* iff its timer variable differs at most one with those at all its neighbors involved in the synchronization. Synchronization is realized by making each timer variable s-consistent and then decrementing it with preserving the s-consistency.

The local neighborhood synchronizer has the following two API for its application. Let $k_{1,2}$ be $\max\{c_1, c_2\} + \min\{f_1, f_2\} + 1$.

Specification 2 ***API***
The following API is available at each process $p \in V$ for the application of the local neighborhood synchronizer:

(i) ***start_synch_NS****: when this function call is executed at process p, it starts the synchronization involving $k_{1,2}$-neighbors of p. These processes decrements their timer variables from $(r_1 + r_2)$ to 0 with keeping s-consistency and their timer variables take 0 in $O(r_1 + r_2)$ rounds.*

(ii) ***exec_NS****: when this function call is executed at process p, if p is enabled in the local neighborhood synchronizer, then it executes one of the corresponding actions, and if p decrements t_p, this function call returns true, otherwise false. If p is not enabled, then p does nothing and this function call returns $\perp$.*

Starting from an f-faulty configuration ($f \leq f_{1,2}$), in the composite protocol, the variables of P_1, P_2, and the local neighborhood synchronizer can be corrupted. In this case, the local

neighborhood synchronizer starts its own recovery actions. However, the source protocols are executed according to the value of the timer variables. Thus, during the recovery actions, the local neighborhood synchronizer should provide correct values of timer variables and synchronization of timer variables at faulty processes.

Synchronization radius is the maximum distance between any faulty process and a process involved in the synchronization caused by the faulty process. To keep the spatial containment of the source protocols, the synchronization radius should be smaller than or equals to $k_{1,2}$. To keep the temporal containment of the source protocols, the local neighborhood synchronizer makes these processes synchronize at most $(r_1 + r_2)$ rounds.

In an f-faulty configuration ($f \leq f_{1,2}$), $t_p = 0$ always holds at correct process p because the variables of correct processes were not corrupted by the fault. So, correct processes are synchronized for (r_1+r_2) rounds by setting their timer variables (r_1+r_2) and then decrementing it. However, it is difficult to make faulty processes decrement their timer variables from $(r_1 + r_2)$ because of corruption of timer variables. From Assumption 3, when a faulty process p is surrounded by other faulty processes, it cannot determine whether it is correct or not. If the value of timer variables at p and all $q \in N_p$ seem to be consistent (i.e. synchronized) with values smaller than $(r_1 + r_2)$, p may start to decrement t_p from the value though it is corrupted by a fault. In the composite protocol, this causes faulty processes to execute P_2 before P_1 recovers. However, as mentioned in the beginning of Section 3.3, this does not a problem for the composition. What is important is that starting from an f-faulty configuration, correct processes always count down their timer variables from $(r_1 + r_2)$.

Specification 3 ***Synchronization***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), the local neighborhood synchronizer provides the following five properties:

(i) ***Spatial containment****: synchronization radius is smaller than or equals to $k_{1,2}$.*

(ii) ***Temporal containment****: the local neighborhood synchronizer reaches a legitimate configuration in $O(r_1 + r_2)$ rounds.*

(iii) ***Synchronization****: each processes involved in the synchronization decrements its timer variable with keeping s-consistency.*

(iv) ***Correct countdown****: if a correct process counts down its timer variable, the timer variable first takes $(r_1 + r_2)$.*

(v) ***Initialization****: if start_synch_NSis executed at process p, then each process in $N_p^{k_{1,2}}$ are involved in the synchronization started by p.*

3.3.2 Composition Protocol *RWFC-LNS*

Our composition framework *RWFC-LNS* (*Fault-containing Composition with the Local Neighborhood Synchronizer*) is shown in Protocol 3.3.1.

Process p executes the guarded actions of the local neighborhood synchronizer by executing exec_NS, and whenever it decrements t_p, p executes the source protocols by executing the procedure $A(t_p)$ that determines which source protocol is executed at p. If $r_2 \leq t_p < r_1 + r_2$ p executes just P_1, otherwise P_2.

When p finds inconsistency in P_1 or P_2 ($cons_p(P_1) = false$ or $cons_p(P_2) = false$ holds at p), there exists a faulty process in $N_p \cup \{p\}$. If there exists a process that is not involved in the synchronization in its neighbors and p ($\{\exists q \in N_p \cup \{p\} : t_q = 0\}$), p executes start_synch_NS and initiates the synchronization of its $k_{1,2}$-neighbors. Then, p and its $k_{1,2}$-neighbors execute P_1 for r_1 rounds. After that, they executes P_2 on the correct input from P_1 and P_2 reaches the legitimate configuration with its fault-containment property.

Protocol 3.3.1 *RWFC-LNS* for $(P_1 * P_2)$

Procedure $A(t_p)$ **for process** p
  if$(r_2 \leq t_p < r_1 + r_2)$ **then execute** P_1
  else execute P_2
Action for process p
  $true \longrightarrow$
  if(exec_NS = ***true*****)** **then** $A(t_p)$**;**
  if$(\neg cons_p(P_1) \vee \neg cons_p(P_2)) \wedge (\exists q \in N_p \cup \{p\} : t_q = 0)$
    then start_synch_NS

3.3.3 Correctness Proof of *RWFC-LNS*

First, we show the stabilization of *RWFC-LNS* (Lemma 1) and then, we show the $f_{1,2}$-fault-containment of *RWFC-LNS*(Theorem 1).

Lemma 1 *Starting from an arbitrary initial configuration, RWFC-LNS eventually reaches the legitimate configuration.*

Proof. Starting from an arbitrary initial configuration, the local neighborhood synchronizer eventually reaches its legitimate configuration (Specification 1). After that, if P_1 is not in a legitimate configuration, then there exists at least one process p that evaluates $cons_p(P_1)$ to

false. Then, process p executes start_synch_NS and initiates the synchronization and p can execute P_1 by executing $A(t_p)$ when it decrements t_p. After the neighborhood synchronizer reaches a legitimate configuration, if P_1 has not reached its legitimate configuration, there exists at least one process that initiates the synchronization by executing start_synch_NS. Until P_1 reaches its legitimate configuration, P_1 is executes in this way. After P_1 reaches a legitimate configuration, if P_2 is not in a legitimate configuration, then there exists at least one process q that evaluates $cons_q(P_2)$ to *false*. Then, process q executes start_synch_NS and initiates the synchronization and q can execute P_2 by executing $A(t_q)$ when it decrements t_q. In the same way as the stabilization of P_1, P_2 eventually reaches its legitimate configuration. Consequently, *RWFC-LNS* eventually reaches a legitimate configuration. □

Then, we have the following theorem.

Theorem 1 *RWFC-LNS provides a* $\min\{f_1, f_2\}$*-fault-containing protocol* $(P_1 * P_2)$ *for* f_1*-fault-containing protocol* P_1 *and* f_2*-fault-containing protocol* P_2*. The contamination radius of the obtained protocol is* $O(\max\{c_1, c_2\} + \min\{f_1, f_2\})$*. The recovery time of the obtained protocol is* $O(r_1 + r_2)$*.*

Proof. From Lemma 1, *RWFC-LNS* is self-stabilizing. In the following, we present the $f_{1,2}$-fault-containment of *RWFC-LNS*.

In an f-faulty configuration ($f \leq f_{1,2}$), for each faulty process p, there exists at least one faulty process $q \in N_p^{\min\{f_1-1, f_2-1\}}$ that is neighboring a correct process. Let this correct process be r. In an f-faulty configuration, t_r takes 0 because it is not corrupted by the fault. Also, from Assumption 3, in an f-faulty configuration, at least one of the following predicates is evaluated to *false*: $cons_q(P_1)$, $cons_q(P_2)$, $cons_r(P_1)$, and $cons_r(P_2)$. We have the following two cases:

Case 1: If r finds inconsistency in the source protocols ($cons_r(P_1)$ or $cons_r(P_2)$ is evaluated to *false*), r executes start_synch_NS because t_r takes 0 in the f-faulty configuration. After that, from the initialization property in Specification 3, each $s \in N_r^{k_{1,2}}$ is involved in the synchronization. From $N_p^{\max\{c_1,c_2\}} \subset N_r^{k_{1,2}}$, each process $s \in N_p^{\max\{c_1,c_2\}}$ counts down t_s from $(r_1 + r_2)$ to 0.

Case 2: If r does not find inconsistency in the source protocols ($cons_r(P_1)$ and $cons_r(P_2)$ are evaluated to *true*), $cons_q(P_1)$ or $cons_q(P_2)$ is evaluated to *false* at q. In this case, q executes start_synch_NS because t_r takes 0 in the f-faulty configuration. After that, from the initialization property in Specification 3, each $s \in N_q^{k_{1,2}}$ is involved in the synchronization. From $N_p^{\max\{c_1,c_2\}} \subset N_r^{k_{1,2}}$, each process $s \in N_p^{\max\{c_1,c_2\}}$ counts down t_s from $(r_1 + r_2)$ to 0.

Thus, in both cases, all processes in $N_p^{max\{c_1,c_2\}}$ are involved in the synchronization. Once a process is involved in the synchronization, it decrements its timer variable from $(r_1 + r_2)$ to 0.

For the first r_1 rounds, the process executes only P_1 and for the latter r_2 rounds, it executes only P_2. Thus, each process in $N_p^{max\{c_1,c_2\}}$ executes P_2 on the correct input from P_1.

A faulty processes can execute P_2 before P_1 recovers when it is surrounded by other faulty processes and the timer variables happen to be consistent. We can treat the resulting state from the execution of P_2 as one that is obtained by the original fault as long as no correct process executes P_2 before the recovery of P_1, i.e. no correct process is contaminated in P_2 by this execution of P_2 at faulty processes. The correct countdown property in Specification 3 guarantees that each correct process starts to decrement its timer variable from (r_1+r_2). So, each correct process executes P_1 for the first r_1 rounds and P_1 reaches the legitimate configuration. After that, it executes P_2 on the correct input from P_1 for the remaining r_2 rounds. Hence, even if faulty processes executes P_2 before P_1 recovers, it does not damage the fault-containment of whole composite protocol.

After one round, no process executes start_synch_NS because there exists no faulty process neighboring a correct process with a timer variable of value 0. The synchronization takes $O(r_1 + r_2)$ rounds to terminate. Thus, the overall recovery time is $O(r_1 + r_2)$.

Because start_synch_NS is executed just at faulty processes and some correct processes neighboring a faulty process and other correct processes never execute start_synch_NS, the contamination radius of the composite protocol is $O(\max\{c_1, c_2\} + \min\{f_1, f_2\})$.

Before the synchronization is initiated by some process that detects inconsistency in the source protocols, the local neighborhood synchronizer may start its own recovery actions because of the corruption on its variables. However, the contamination is contained in $k_{1,2}$ radius for each faulty process (spatial containment property in Specification 3) and the contamination ends in $O(r_1 + r_2)$ rounds (temporal containment property in Specification 3). Thus, the contamination is contained.

Hence, Protocol 3.3.1 provides an $f_{1,2}$-fault-containing protocol $(P_1 * P_2)$. □

3.4 Local Neighborhood Synchronizer

In this section, we present an implementation of the local neighborhood synchronizer in Section 3.4.1 and prove that *LNS* satisfies the Specification 1, Specification 2, and Specification 3 in Section 3.4.2 and Section 3.4.3.

3.4.1 Protocol *LNS*

For any given M and k, protocol *LNS* provides the synchronization of M rounds among k-neighbors of a process when the process gives the initialization signal by start_synch_NS. The

synchronization consists of three phases. In the first phase, the shortest path tree (SPT) of depth k and rooted at the process is constructed so that all the k-neighbors of the process are involved into the synchronization. Then, in the second phase, the synchronized countdown of timer variables takes place among these processes. In the third phase, the shortest path tree is released from the root to the leaves.

Protocol *LNS* is shown in Protocol 3.4.1. Each process p has four variables, t_p, d_p, $Predicate_p^{init}$ and ret_p: t_p is the timer variable, d_p is the depth variable which is used to construct the SPT, and $Predicate_p^{init}$ and ret_p are boolean variables that are used to implement the API defined in Specification 2. Protocol *LNS* is self-stabilizing and in a legitimate configuration, $(t_p = 0 \wedge d_p = 0 \wedge Predicate_p^{init} = \mathit{false} \wedge ret_p = \mathit{false})$ holds at each $p \in V$.

API in Specification 2 is implemented as follows: When the application of *LNS* executes start_synch_NS at process p, we assume it changes $Predicate_p^{init}$ from *false* to *true*. If the predicate $Predicate_p^{init} = \mathit{true}$ holds at p, *LNS* changes the value of t_p to M and d_p to k. After p executes *LNS* (and $(t_p = M \wedge d_p = k)$ holds), the application of *LNS* should change $Predicate_p^{init}$ from *true* to *false*. The return value of exec_NS is implemented with ret_p that returns *true* iff the execution of *LNS* decrements t_p, otherwise *false*.

We call process p a *root* iff the value of d_p is locally maximum among its direct neighbors. For each root process p, each non-root process $q \in N_p^{d_p}$ constructs the SPT rooted at p by setting $d_q = d_p - dist(p, q)$ where $dist(p, q)$ denotes the distance between p and q. The parent(s) of q is any neighbor $r \in N_q$ where $d_r = d_q + 1$. A process $s \in N_q$ is a child of q iff $d_s = d_q - 1$.

Protocol *LNS* consists of seven guarded actions, $S_1, \cdots, S_7$. Starting from an arbitrary initial configuration, *LNS* allows process p, where $t_p = M$ holds in the initial configuration, to become a root and to construct the SPT rooted at itself. When start_synch_NS is executed at process p ($Predicate_p^{init} = \mathit{true}$), p becomes a root of the SPT of depth k by executing S_1. When process $q \in N_p$ finds that the value of t_q is smaller than M ($R_p(1)$) and it is neighboring a process p with $t_p = M$ ($R_p(2)$), q executes S_2 and change the value of t_q to M and the value of d_q to $d_p - 1$. A non-root process $q \in N_p^{d_p}$ is involved in the SPT tree rooted at p by executing S_2 (and S_3 if necessary) and setting $t_q = M$ and $d_q = d_p - dist(q, p)$. Note that there may be multiple root processes and each non-root process is involved in the SPT rooted at the nearest root process. After $(t_q = M \wedge d_q = d_p - dist(p, q))$ holds at q and all its neighbors get involved in the shortest path tree, q goes into the second phase.

In the second phase, q decrements t_q by executing S_4 iff t_q is synchronized with all its neighbors ($OK_t_q = \mathit{true}$) and all its neighbors have involved in the SPT ($OK_d_q = \mathit{true}$). The guard dec_q make the execution of S_4 at q's parent precede the execution of S_4 at q when the value of the timer value takes the same value ($D_q(1)$ and $D_q(2)$). So, the execution of S_4 starts

Protocol 3.4.1 *LNS*

Local variables at process p

t_p: timer variable that takes a value in $[0..M]$

d_p: depth variable that takes a value in $[0..k]$

For API at process p

$Predicate_p^{init}$: an input variable that takes a boolean value

ret_p: an output variable to the application that takes a boolean value

Predicates at process p

$OK_d_p \equiv \{\forall q \in N_p : |d_p - d_q| \leq 1\} \wedge \{(\exists q \in N_p : d_p = d_q - 1) \vee (\forall q \in N_p : d_p \geq d_q)\}$

$OK_t_p \equiv \{(d_p > 0) \wedge (\forall q \in N_p : |t_p - t_q| \leq 1)\} \vee$
$\{(d_p = 0) \wedge (\forall q \in N_p : (|t_p - t_q| \leq 1 \wedge d_p = d_q + 1) \vee (d_q = 0))\}$

$raise_p \equiv R_p(1) \wedge R_p(2)$

$R_p(1) \equiv (t_p \neq M)$

$R_p(2) \equiv \{\exists q \in N_p : (t_q = M) \wedge (d_q > 0) \wedge \neg((t_p = M - 1) \wedge (d_p = d_q + 1))\}$

$maxd_p \equiv M_p(1) \wedge M_p(2) \wedge M_p(3)$

$M_p(1) \equiv (\max_{q \in N_p}\{d_q\} \neq 0) \wedge (d_p < \max_{q \in N_p}\{d_q\} - 1)$

$M_p(2) \equiv (d_p \neq k) \wedge \neg(\forall q \in N_p : d_p > d_q)$

$M_p(3) \equiv \{\forall q \in N_p \cup \{p\} : t_q = M \vee (t_q = 0 \wedge d_q = 0)\}$

$dec_p \equiv OK_d_p \wedge OK_t_p \wedge D_p(1) \wedge D_p(2)$

$D_p(1) \equiv (t_p > 0) \wedge (\forall q \in N_p : t_p \geq t_q)$

$D_p(2) \equiv (\forall q \in N_p : t_p = t_q \rightarrow d_p \geq d_q)$

$clrd_p \equiv C_p(1) \wedge C_p(2)$

$C_p(1) \equiv (t_p = 0) \wedge (\forall q \in N_p : t_q = 0)$

$C_p(2) \equiv (d_p > 0) \wedge \{\forall q \in N_p : d_p \geq d_q \vee d_q = 0\}$

$rset_p \equiv \neg raise_p \wedge \neg maxd_p \wedge \neg dec_p \wedge \neg clrd_p \wedge$
$\neg(t_p = M) \wedge \neg(OK_d_p \wedge OK_t_p) \wedge \neg(t_p = 0 \wedge d_p = 0)$

Actions for process p

S_1 $Predicate_p^{init} \longrightarrow t_p = M;\ d_p = k$

S_2 $\neg Predicate_p^{init} \wedge raise_p \longrightarrow t_p = M;\ d_p = \max_{q \in N_p \wedge t_q = M}\{d_q\} - 1$

S_3 $\neg Predicate_p^{init} \wedge maxd_p \longrightarrow d_p = \max_{q \in N_p}\{d_q\} - 1$

S_4 $\neg Predicate_p^{init} \wedge dec_p \wedge ret_p = false \longrightarrow t_p = t_p - 1;\ ret_p = true$

S_5 $\neg Predicate_p^{init} \wedge clrd_p \longrightarrow d_p = 0$

S_6 $\neg Predicate_p^{init} \wedge rset_p \longrightarrow d_p = 0;\ t_p = 0$

S_7 $ret_p = true \longrightarrow ret_p = false$

from the root process and spreads to the leaves in a top-down fashion. When q executes S_4, ret_q is changed from *false* to *true* and exec_NS returns *true*. After that, q reset ret_q to *false* by executing S_7 before it executes S_4 again. Note that the first phase and the second phase can be executed in parallel but each process involved in the SPT start the second phase after all its neighbors have finished the first phase.

In the third phase, after all the neighbors finish the countdown ($C_q(1)$), q executes S_5 and sets $d_q = 0$. The execution of S_5 also starts from the root and spreads to leaves ($C_q(2)$), and the SPT is released. Eventually, the third phase ends and $t_q = 0 \wedge d_q = 0$ holds at each $q \in V$.

During the stabilization, if process p finds that the guards $raise_p$, $maxd_p$, dec_p, $clrd_p$ are all *false* or it is not waiting its neighbors to attend an SPT ($\neg(t_p = M)$) or it is not waiting its neighbors to decrement their timer variables ($\neg(OK_t_p \wedge OK_d_p)$), it resets t_p and d_p by executing S_6. This behavior does not prevent the progress of above three phases.

3.4.2 Correctness Proof: Stabilization of *LNS*

In this section, we show the stabilization of *LNS* defined in Specification 1. We first show the behavior of *LNS* when *LNS* is started from an arbitrary initial configuration. In the following, we assume that $Predicate_p^{init}$ is *false* and the application of *LNS* does not affect the behavior of *LNS*. So, no process executes S_1. We first focus on the SPTs constructed during the execution and then we show the stabilization of the whole system with Lemma 7.

We first show that if the system is not in a legitimate configuration, there is at least one process that executes a guarded action of *LNS*. Then, we show the stabilization of *LNS*. Starting from an arbitrary initial configuration, the set of root processes of SPTs is determined by the states of processes in the initial configuration. Process p can become a root process if $t_p = M$ holds in the initial configuration (Lemma 3). For process p, let $R_p^{N^k}$ be the set of processes such that $q \in N_p^k \cup \{p\}$ and $t_q = M$ holds in the initial configuration. So, $R_p^{N^k}$ is the set of possible root processes in N_p^k. Each non-root process p is involved in the SPT rooted at the nearest root process in $R_p^{N^k}$ (Lemma 4). Then, p decrements t_p and its neighbors also decrements their timer variables (Lemma 5). Finally, p decrements t_p from M with keeping s-consistency and t_p eventually reaches 0 (Lemma 6).

Lemma 2 *For any configuration, if the configuration is not legitimate, at least one process has an enabled guard and executes the corresponding action.*

Proof. If a configuration is not legitimate, there exists at least one process $p \in V$ where $\neg(t_p = 0 \wedge d_p = 0 \wedge ret_p = false)$ holds. If $ret_p = true$ holds, then p executes S_7 and changes

ret_p to *false*. In the following, we focus on the case where $(\neg(t_p = 0 \wedge d_p = 0) \wedge (ret_p = false))$ holds at p.

For contradiction, consider the case where all the guards of $S_2, \cdots, S_6$ are evaluated to *false* at all $q \in V$ in an illegitimate configuration. (We do not consider S_1 because in this section, we assume $Predicate_q^{init} = false$ always holds.) Thus, the following predicate holds at each $q \in V$:

$$\begin{aligned}
& raise_q \vee maxd_q \vee dec_q \vee clrd_q \vee \\
& \{\neg raise_q \wedge \neg maxd_q \wedge \neg dec_q \wedge \neg clrd_q \\
& \wedge \neg(t_q = M) \wedge \neg(OK_d_q \wedge OK_t_q) \wedge \neg(t_q = 0 \wedge d_q = 0)\} \\
=& (raise_q \vee maxd_q \vee dec_q \vee clrd_q) \vee \\
& \{\neg(raise_q \vee maxd_q \vee dec_q \vee clrd_q) \\
& \wedge \neg(t_q = M) \wedge \neg(OK_d_q \wedge OK_t_q) \wedge \neg(t_q = 0 \wedge d_q = 0)\} \\
=& false.
\end{aligned}$$

Because we assume $S_2, \cdots, S_6$ are evaluated to *false* at q, $(raise_q \vee maxd_q \vee dec_q \vee clrd_q) = false$ holds. Thus, we obtain

$$\begin{aligned}
& false \vee \{true \wedge \neg(t_q = M) \wedge \neg(OK_d_q \wedge OK_t_q) \wedge \neg(t_q = 0 \wedge d_q = 0)\} \\
=& false
\end{aligned}$$

So, $(\neg(t_q = M) \wedge \neg(OK_d_q \wedge OK_t_q) \wedge \neg(t_q = 0 \wedge d_q = 0)) = false$ holds at each $q \in V$.

By assumption, $\neg(t_p = 0 \wedge d_p = 0)$ holds at p. By above discussion, $(t_p = M)$ or $(OK_t_p \wedge OK_d_p)$ is *true* at p. We have the following two cases.

(i) $t_p = M$: In this case, we have the following two cases:

(a) $t_q = M$ holds for all $q \in N_p$: In this case, we have the following two cases:

- $|d_p - d_q| \leq 1$ holds for all $q \in N_p$: If $d_p \geq d_q$ for all $q \in N_p$, then p evaluates dec_p to *true*. Otherwise, there exists process $r \in N_p$ such that $(d_r > d_p)$ and $(t_r = M)$ hold and d_r evaluates dec_r to *true*.
- $|d_p - d_q| > 1$ holds for some $q \in N_p$: thus, q evaluates $maxd_q$ to *true*.

(b) $t_q < M$ holds for some process $q \in N_p$: At such process q, $(OK_d_q \wedge OK_t_q)$ or $(t_q = 0 \wedge d_q = 0)$ holds. If $(OK_d_q \wedge OK_t_q)$ holds at q, then $(t_q = M-1)$ and $(|d_p - d_q| \leq 1)$ hold and p evaluates dec_p to *true*. If $(t_q = 0 \wedge d_q = 0)$ holds at q, then q evaluates $maxd_q$ to *true*.

(ii) $OK_d_p \wedge OK_t_p$: In this case, we have the following two cases:

(a) $(\forall q \in N_p : t_p \geq t_q)$ holds at p: If $(\forall q \in N_p : t_p = t_q \rightarrow d_p \geq d_q)$ holds at p, then dec_p

is evaluated to *true*. Otherwise, there exists process $q \in N_p$ such that $(t_p = t_q \wedge d_p < d_q)$ holds. In this case, dec_q is evaluated to *true* at q.
(b) $(\exists q \in N_p : t_p < t_q)$ holds at p: Let q be the process where $(t_q > t_p)$ holds. Because OK_d_p and OK_t_p holds at p, $t_q = t_p{+}1$. At process q, $((t_q = M) \vee (OK_d_q \wedge OK_t_q) \vee (t_p = 0 \wedge d_p = 0))$ also holds. If $(t_q = M)$ holds at q, then q follows case (i). If $(OK_td_p \wedge OK_t_q)$ holds at q, q evaluates dec_q to *true*. Because t_q is larger than $t_p(\geq 0)$, $(t_p = 0 \wedge d_p = 0)$ does not hold at q.

Hence, there is contradiction and there exists at least one process that evaluates one of the guards of $S_2, \cdots, S_6$ to *true* and executes the corresponding action. □

Lemma 3 *Starting from an arbitrary initial configuration, if $t_p < M$ holds in the initial configuration at process p, the SPT rooted at p is not constructed during any execution.*

Proof. For contradiction, consider the case such that $t_p < M$ holds at process p in the initial configuration and the SPT rooted at p is constructed during the execution. The values of depth variables are changed by S_2, S_3, S_5, and S_6. (Note that process p never executes S_1 because we assume $Predicate_p^{init}$ is *false* in this section.) To construct the SPT rooted at p, the value of d_p remains larger than d_q for each $q \in N_p$ until $(d_p = 0 \wedge t_p = 0)$ holds at p and each $q \in N_p$ should change the value of d_q to $(d_p - 1)$.

During the execution, p can execute S_2, S_3, S_5, and S_6 to change the value of d_p. By the execution of S_5 or S_6, d_p takes zero. By the execution of S_2 or S_3, the value of d_p is changed to $d_r - 1$ for a process $r \in N_p$. So, by the execution of *LNS*, d_p cannot take a larger value than $d_{r'}$ for any $r' \in N_p$.

For each $q \in N_p$, d_q cannot take $(d_p - 1)$ by executing S_2 or S_3 because d_p is smaller than M. Clearly, the execution of S_5 or S_6 cannot make d_q to take $(d_p - 1)$.

Hence, the SPT rooted at p never constructed during the execution. □

From Lemma 3, the possible root processes are determined by the initial configuration. If $t_p = M$ holds at process p in the initial configuration, d_q take the value $(d_p - 1)$ for each $q \in N_p$ as long as there is no process r such that $(d_r - dist(r, q))$ is larger than $(d_p - 1)$. In this case, the SPT rooted at p is constructed during the execution. If such process r exists, then p may be involved in the SPT rooted at r even if $t_p = M$ holds in the initial configuration. In this case, all the descendants of p in p's SPT is involved in r's SPT.

Lemma 4 *Starting from an arbitrary initial configuration, for non-root process p, if $t_q = M$ holds at some process $q \in N_p^k$ in the initial configuration, d_p eventually takes the value of $max_{q \in R_p^{N^k}}\{d_q - dist(p, q)\}$.*

Proof. Let p be a non-root process such that $R_p^{N^k} \neq \emptyset$ holds in the initial configuration. We show by induction that d_p eventually takes $(d_q - dist(p,q))$ for process $q \in R_p^{N^k}$ that maximize value of $(d_q - dist(p,q))$. Clearly, for each process r on the shortest path(s) from q to p, q maximize the value of $(d_q - dist(r,q))$.

Starting from an arbitrary initial configuration, process $r \in N_q$ eventually executes S_2 or S_3 and d_r takes $(d_q - 1)$ because q is the nearest root process. Until d_r takes $(d_p - 1)$ for each $r \in N_q$, q cannot decrement t_q (from S_4). (If q starts to decrement t_q, t_q can reach 0 and d_q may take 0 by executing S_5 before the SPT is constructed.) Thus, eventually $(d_r = d_q - 1 \wedge t_r = M)$ holds at each $r \in N_q$.

Let process r' and r'' be on the shortest path from q to p and $dist(q,r'') = dist(q,r') + 1$. Let $(d_{r'} = d_q - dist(q,r') \wedge t_{r'} = M)$ eventually hold during the execution. Then, after that, r'' executes S_2 or S_3 and $t_{r''}$ takes $d_{r'} - 1 = d_q - dist(q,r') = d_q - dist(q,r'')$. Until $d_{r''}$ takes $(d_{r'} - 1)$, r' cannot decrement $t_{r'}$ (from S_4). Thus, eventually $(d_{r''} = d_q - dist(q,r'') \wedge t_{r''} = M)$ holds at r''.

Thus, eventually, d_p takes $d_q - dist(p,q)$. □

Then, we focus the execution of S_4 at each process to show that all the processes involved in an SPT can keep on decrementing its timer variable from M to 0. Consider the case such that p and $q \in N_p$ evaluates the guard of S_4 to *true* in a configuration. If p executes S_4 first, the execution of S_4 at p should not prevent the execution of S_4 at q, i.e. the evaluation of the guard of S_4 should remain *true*. When p and q decrements their timer variables at the same time, it does not matter the countdown.

Lemma 5 *The execution of S_4 at any process p does not change dec_q at any $q \in N_p$ from true to false.*

Proof. Let $dec_p = true$ hold at process p and $dec_q = true$ hold at process $q \in N_p$. Thus, dec_p and dec_q, $D_p(1)$, $D_p(2)$, $D_q(1)$, and $D_q(2)$ are evaluated to *true*. From $D_p(1)$ and $D_q(1)$, we have $t_p = t_p$. From $D_p(2)$ and $D_q(2)$, we have $d_p = d_q$.

Consider the case where only p executes S_4. After the execution of S_4 at p, we have $t_q = t_p - 1$. Because S_4 just changes the value of t_p, the evaluation of OK_t_q, $D_q(1)$, and $D_q(2)$ are not changed by the execution of S_4 at p. So, the execution of S_4 at process p does not change dec_q at any $q \in N_p$ from *true* to *false*. □

Lemma 6 *Each process p involved in an SPT rooted at a root process decrements t_p from M to 0 with keeping the s-consistency of timer variables.*

Proof. Let p be a process such that $R_p^{N-k} \neq \emptyset$ holds in the initial configuration. From Lemma 4, d_p eventually takes $max_{q \in R_p^{N^k}}\{d_p - dist(p,q)\}$ and p is involved in an SPT correctly. Because

S_4 forces each process to wait until all its neighbors keep the consistent depth value and S_4 allows the processes with higher depth values to decrement the timer variable, p cannot execute S_4 until d_p keeps the correct value. Thus, $(t_p = M \wedge d_p = max_{q \in R_p^{N^k}}\{d_p - dist(p,q)\}$ eventually holds at p and all the neighbors. We will show by induction on t_p that process p decrements t_p from M to 0 with keeping the s-consistency.

From Lemma 5, the execution of S_4 at p does not change dec_q at each $q \in N_p$ from *true* to *false*. Thus, after p executes S_4 and the value of t_p changes from M to $(M-1)$, its neighbors eventually executes S_4, and after that $dec_p = true$ holds again.

Let t_p be $(M - \ell)$ and $dec_p = true$ hold at p. From Lemma 5, the execution of S_4 at p does not change dec_q at each $q \in N_p$ from *true* to *false*. Thus, after the execution of S_4 at p, its neighbors eventually executes S_4, and after that $dec_p = true$ holds again.

Consequently, p execute S_4 and decrement t_p with keeping s-consistency until t_p reaches 0. □

Finally, we show that the system eventually reaches a legitimate configuration. Note that even when $\neg(t_p = 0 \wedge d_p = 0 \wedge ret_p = false)$ holds at process p in the initial configuration, it is possible that there is no root process in N_p^k during the stabilization.

Lemma 7 *Starting from an arbitrary initial configuration, the system reaches a configuration where* $(t_p = 0 \wedge d_p = 0 \wedge ret_p = false)$ *holds at each* $p \in V$.

Proof. For a configuration, let $d_{max}(V)$ ($t_{max}(V)$, respectively) be the maximum value of d_p (t_p, respectively) for all $p \in V$. We show that starting from an arbitrary initial configuration, $d_{max}(V)$ and $t_{max}(V)$ eventually reaches 0.

We first show the outline of the progress of stabilization. The set of root processes are defined by the initial configuration (Lemma 3). All the SPTs rooted at these processes are eventually constructed by executing S_2 and S_3 (Lemma 4). Let σ_{SPT} be the configuration such that after σ_{SPT}, no process executes S_2 and S_3. The system eventually reaches σ_{SPT} because the execution of *LNS* does not make new root process(es). For each root process, once the SPT is constructed, each process p involved in the SPT decrements t_p and eventually t_p takes 0 by executing S_4 (Lemma 5). Let σ_{dec} be the configuration such that after σ_{dec}, no process executes S_4. The system eventually reaches σ_{dec} because M is a finite value. After the system reaches σ_{dec}, no process executes S_6 because no timer variable changes its value, and no depth variable changes its value to take a larger value. After p and all its neighbors have finished the countdown, they execute S_5. Eventually, all SPTs are removed. Let σ_{clr} be the configuration such that after σ_{clr}, no process executes S_5. The system eventually reaches σ_{clr}.

We show the decrement of $t_{max}(V)$ and $d_{max}(V)$ during the execution. The system first reaches σ_{SPT}, then σ_{dec}, and finally σ_{clr}. Until the system reaches σ_{SPT}, $t_{max}(V)$ is smaller

than or equals to M and after σ_{SPT}, $t_{max}(V)$ decreases monotonically. Let t_p at process p takes $t_{max}(V)(> 0)$ in a configuration after σ_{SPT}. Because no process executes S_2 after SPTs are constructed, OK_d_p holds. Also, OK_t_p holds at p because after σ_{SPT}, p changes the value of d_p by executing S_4 and S_6. The execution of S_4 preserves OK_t_p and because $t_p = t_{max}(V)$, p has not executed S_6. Because $t_p = t_{max}(V)$, $D_p(1)$ holds at p. And without loss of generality, we can assume $d_p \geq d_q$ when $t_p = t_q = t_{max}(V)$ holds for $q \in N_p$. So, $dec_p = \mathit{true}$ holds at p and p decrements t_p by executing S_4. After the system reaches σ_{dec}, $t_{max}(V)$ remains 0 because no process executes S_2 thereafter.

Until the system reaches σ_{dec}, $d_{max}(V)$ is smaller than or equals to k and after σ_{dec}, $d_{max}(V)$ decreases monotonically. Let d_p takes $d_{max}(V)(> 0)$ in a configuration after σ_{dec}. Because $(t_q = 0)$ holds each process $q \in V$ after σ_{dec}, $C_p(1)$ holds at p. Because $d_p = d_{max}(V)$, $C_p(2)$ holds at p. So, $clrd_p$ is evaluated to *true* at p and p decrements d_p. After the system reaches σ_{clr}, $d_{max}(V)$ remains zero because no process executes S_2 or S_3 thereafter.

After the system reaches σ_{clr}, $ret_p = \mathit{false}$ holds at each process in one round by the execution of S_7.

Consequently, the system eventually reaches a configuration where $(t_p = 0 \wedge d_p = 0 \wedge ret_p = \mathit{false})$ holds for each process $p \in V$. □

Lemma 8 *Starting from an arbitrary initial configuration, the system reaches a legitimate configuration in $O(k + M)$ rounds.*

Proof. From Lemma 6 and Lemma 7, non-root process p is involved in an SPT or reset t_p and d_p. If p is involved in an SPT, it decrements t_p from M to 0 and after that p changes d_p to 0.

From Lemma 3, root processes are determined by the initial configuration. From Lemma 4, the SPTs rooted at root processes are constructed. The SPTs construction takes at most k rounds because the depth of each SPT is at most k. From Lemma 5, once SPTs are constructed, the synchronized countdown of tree processes takes place. It takes M rounds for all the tree processes count down from M to 0.

Each process p executes S_5 when all its neighbors has finished decrementing their timer variables (thus, for each $q \in N_p$, $t_p = 0$). From $clrd_p$, p can execute S_5 after all the neighbors with higher depth values execute S_5. Thus, the execution of S_5 starts from the root process to the leaves of its SPT. It takes at most k rounds for all the tree processes to execute S_5 and $(t_p = 0 \wedge d_p = 0)$ holds at each process $p \in V$. It takes at most one round for each process to chance ret_p to *false* (if necessary) by executing S_7.

Once $(t_p = 0 \wedge d_p = 0 \wedge ret_p = \mathit{false})$ holds at each process $p \in V$, no process is enabled and each process p does not change the values of its local variables, t_p, d_p, and ret_p. Hence, the system reaches a legitimate configuration and the stabilization time of *LNS* is $O(k + M)$. □

From Lemma 8, the following theorem immediately holds.

Theorem 2 *Protocol LNS is self-stabilizing.*

3.4.3 Correctness Proof: Synchronization of *LNS*

In this section, we show that *LNS* provides the API defined in Specification 2 and the five specifications defined for neighborhood synchronizer in Specification 3 for $M = (r_1 + r_2)$ and $k = k_{1,2}$.

LNS satisfies the spatial containment property because only faulty processes can become root processes, and *LNS* just involves their $k_{1,2}$-neighbors into the synchronization (Lemma 9). From Lemma 8 in Section 3.4.2, the synchronization of *LNS* takes $O(k + M)$ rounds. From definition, $k + M = (r_1 + r_2) + \max\{c_1, c_2\} + \min\{f_1, f_2\} + 1$ and r_1 (r_2, respectively) is larger than f_1 and c_1 (f_2 and c_2, respectively). So, $O(k + M)$ equals to $O(r_1 + r_2)$ and *LNS* satisfies the temporal containment property. The synchronization property holds directly from Lemma 5 in Section 3.4.2. The correct countdown property is satisfied because in any execution starting from a target faulty configuration, each correct process first sets its timer variable $(r_1 + r_2)$ (Lemma 10). The initialization property is the special case of Lemma 4 in Section 3.4.2 such that the timer variable takes $(r_1 + r_2)$ and the depth variable takes $k_{1,2}$ at the root process (Lemma 11).

Lemma 9 ***Spatial containment***
Starting from an arbitrary initial configuration, only faulty processes can become root processes and construct SPTs. Hence, the spatial containment property in Specification 3 is satisfied.

Proof. From Lemma 3, p can become a root process only when $t_p = M$ holds in the initial configuration.

At a correct process q, $(t_q = 0 \wedge d_q = 0)$ holds in the initial configuration. Thus, starting from an f-faulty configuration, a correct process never become a root process of an SPT. On the other hand, $(d_p \neq 0 \vee t_p \neq 0)$ holds at a faulty process p. Thus, only faulty processes can become a root process of an SPT.

From Lemma 4, only the $k_{1,2}$-neighbors of each root process are involved in the SPT. Thus, *LNS* satisfies the spatial containment property. □

Lemma 10 ***Correct countdown***
Starting from an faulty configuration, if a correct process p changes its state during the recovery, p first executes S_2 and always countdown from $(r_1 + r_2)$.

Proof. In an f-faulty configuration, $(t_p = 0 \wedge d_p = 0 \wedge ret_p = \mathit{false})$ holds at a correct process p. Thus, $init_p$, $maxd_p$, dec_p, and $clrd_p$ are evaluated to *false*. The only predicate that can be evaluated to *true* at p when $(t_p = 0 \wedge d_p = 0)$ holds is S_2. By executing S_2, t_p takes $(r_1 + r_2)$. □

Lemma 11 ***Initialization***
If process p executes start_synch_NS, then after that all processes in $N_p^{k_{1,2}} \cup \{p\}$ *count down their timer variable from* $(r_1 + r_2)$ *to* 0.

Proof. The execution of start_synch_NS at process p changes $Predicate_p^{init}$ from *false* to *true*. When $Predicate_p^{init}$ is *true*, p can execute only S_1 and S_7. Thus, S_1 becomes a root process and after that, the SPT rooted at p is constructed (Lemma 4). From Lemma 6, each process in $N_p^{k_{1,2}}$ decrements its timer variable from $(r_1 + r_2)$ to 0.

During the recovery, there may be multiple SPTs rooted at different root processes. However, the distance from any root process to any other root process is at most $\min\{f_1, f_2\}$. Thus, the SPTs encounter each other during the SPT construction phase. So, a process involved in SPTs waits its depth variable to take correct value for its nearest root process and after that it starts to decrement its timer variable from M to 0.

Note that if faulty process p is surrounded by other faulty processes, then p may become a root process even if d_p is smaller than k. In this case, just among faulty processes, there may be an SPT such that the depth of the SPT is smaller than k. However, correct processes always attend an SPT of depth k because the nearest root process for a correct process executes start_synch_NS. □

Consequently, we have the following theorem.

Theorem 3 *Starting from an f-faulty configuration, LNS satisfies Specification 3 for* $k = k_{1,2}$ *and* $M = (r_1 + r_2)$.

3.5 Concluding Remarks

In this chapter, we first introduced the *RWFC* strategy for fault-containing composition. Our strategy is to stop the execution of the upper protocol until the lower protocol recovers. We can compose more than two fault-containing protocols with *RWFC* strategy by applying *RWFC* strategy repeatedly to the source protocols. Though the strategy is very simple, it provides significant improvement on composing fault-containing protocols. Furthermore, this framework helps designing new fault-containing protocols, e.g. we can easily built new fault-containing protocols on top of existing fault-containing protocols.

In this chapter, we proposed a fault-containing composition *RWFC-LNS* that utilizes the temporal containment property of fault-containing protocols. The proposed composition technique *RWFC-LNS* stops the upper protocol during the recovery time of the lower protocol and utilizes timers at processes to measure the recovery time of the source protocols. To implement timers, we designed a local neighborhood synchronizer protocol *LNS*.

In *RWFC-LNS*, when some process finds inconsistency in the lower protocol, each process in the contamination radius stops the upper protocol during the recovery time of the lower protocol. Though *LNS* imposes additional communication overhead and time complexity, the overall overhead is bounded by the contamination radius of source protocols. The contamination of the composite protocol is also bounded by the contamination radius of source protocols. Hence, the proposed composition technique preserves the fault-containment property of source protocols. Unfortunately, because the proposed framework stops the upper protocol for the recovery time of the lower protocol, even when the lower protocol has recovered earlier than its recovery time, the upper protocol does not resume immediately.

The notion of local neighborhood synchronizer and the implementation *LNS* are useful in fault-containment and other fault-tolerant distributed protocols. For example, we can improve the spatial containment property of [23] by replacing the global neighborhood synchronizer with our local neighborhood synchronizer. Other applications of the local neighborhood synchronizer remain to be discussed and developed.

Chapter 4

Hierarchical Composition with Spatial Containment

Fault-containing protocols provide temporal containment and/or spatial containment. In Chapter 3, we proposed fault-containing composition *RWFC-LNS* based on temporal containment property of source protocols. In this chapter, we present a fault-containing composition technique based on the spatial containment property of source protocols. The proposed composition technique also improves the recovery scenario of *RWFC-LNS*. In the worst case, *RWFC-LNS* keeps the upper protocol waiting after the lower protocol has recovered. This is because the recovery time of a fault-containing protocol is the maximum (worst) time necessary for the recovery. In this chapter, we improve the recovery scenario by executing the upper protocol as soon as the lower protocol recovers.

The proposed composition technique checks the inconsistency of the lower protocol to detect the recovery of the lower protocol. Generally, in self-stabilizing protocols, each process checks the inconsistency between its neighbors with local predicates. In fault-containing protocols, each process utilizes this local consistency predicate to find faulty processes and if it finds a faulty process, then the process waits for the recovery actions of the faulty process so that the effect of the fault is contained around the faulty process. In this chapter, we utilize this local consistency predicate to detect the recovery of the lower protocol. We call the proposed composition technique *RWFC-IcD* (*RWFC* with the Inconsistency Detector). The *inconsistency range* of a fault-containing protocol guarantees that from a target faulty configuration, there always exists at least one process that finds inconsistency in the inconsistency range from a faulty process. We force each process to check the consistency of the lower protocol among the inconsistency range. When the process finds no inconsistency in the lower protocol, it is guaranteed that the process can execute the upper protocol on a correct input from the lower

protocol. Then, the composition protocol allows the process to execute the upper protocol. Thus, the upper protocol can recover with its fault-containment property and the composite protocol promises fault-containment as a whole.

To implement *RWFC-IcD*, it is necessary to implement the inconsistency detector that allows each process to detect the inconsistency of the lower protocol. We implement the inconsistency detector *IcD* based on an existing *Propagation of Information with Feedback* (PIF) protocol *PIF* in [12], however, *PIF* itself is not fault-containing. Hence, we modified *PIF* to provide spatial and temporal containment property.

This chapter is organized as follows. In Section 4.1, we show conditions on source protocols. In Section 4.2, we first define the specification of the inconsistency detector and then present the composition framework *RWFC-IcD*. The correctness proof of *RWFC-IcD* is also shown in Section 4.2. In Section 4.3, we present an implementation of the inconsistency detector, protocol *IcD* and prove that *IcD* satisfies the specification in Section 4.2. We conclude this chapter with Section 4.4.

4.1 Preliminary

In this chapter, we consider self-stabilization and fault-containment of protocols for non-reactive problems. We follow Definition 5 (fault-containing composition), Remark 1 and Assumption 1 in Section 3.1. We also follow Definition 6 (*RWFC* strategy) and Remark 2 in Section 3.1.

In this chapter, we put some assumptions on the source protocols of fault-containing compositions. We consider a subclass of fault-containing protocols Π such that each f-fault-containing protocol $P \in \Pi$ satisfies Assumption 1, 2, 5, and 6. (Note that Assumption 1 (unique legitimate configuration), and Assumption 2 (legitimacy predicate) are defined in Chapter 3.) Many existing fault-containing protocols satisfy Assumption 1, 2, 5, and 6 [33, 24, 25, 22].

Assumption 5 *In an f'-faulty configuration ($f' \leq f$), if faulty process p is a neighbor of correct process(es), at least one correct process $q \in N_p$ or p itself evaluates $cons_q(P)$ (or $cons_p(P)$) false.*

Many fault-containing protocols satisfies Assumption 5: for a faulty process p and a neighboring correct process q, the predicate $cons_p(P)$ ($cons_q(P)$, respectively) involves the local variables at q (p, respectively). Because p is faulty, there can be some inconsistency between the local state of p and that of q.

Note that if p and all its neighbors are faulty, $cons_p(P) = true$ may hold at p. This is because $cons_p(P)$ involves the local variables at p and its neighbors and the values of these corrupted

variables happen to seem consistent. In this case, p cannot determine whether it is faulty or not.

The *inconsistency range* of P is the maximum (worst) distance from any faulty process to the process q that evaluates $cons_q(P)$ *false* because of the faulty process during the recovery from an f'-faulty configuration ($f' \leq f$).

Assumption 6 *Let k be the inconsistency range of P. Starting from any f'-faulty configuration ($f' \leq f$), for each faulty process p, in every configuration there exists at least one process $q \in N_p^k \cup \{p\}$ such that $cons_q(P)$ is evaluated false until the local variables at p takes the values that they take in the legitimate configuration.*

The upper bound of the inconsistency range of a protocol is obtained by its contamination number or recovery time that are always larger than or equals to the inconsistency range. We can obtain the more accurate value of the inconsistency range by analyzing the behavior of the protocol. In many 1-fault-containing protocols [33, 24, 25, 22], inconsistency range is 1: in theses protocols, in a 1-faulty configuration the faulty process or its neighbors may suspect it is faulty and exchange the local information with neighbors. If a correct process finds the faulty process, the process waits until the faulty process changes its variables.

4.2 Composition Framework

Let P_1 be an f_1-fault-containing protocol and P_2 be an f_2-fault-containing protocol. Our goal is to produce $f_{1,2}$-fault-containing protocol $(P_1 * P_2)$ for $f_{1,2} = \min\{f_1, f_2\}$. In this chapter, we use the notations shown in Table 4.1.

Table 4.1: Notations for the source protocols and the composite protocol (*RWFC-IcD*)

protocol	number of maximum faults	recovery time	contamination number	inconsistency range
P_1	f_1	t_1	c_1	k_1
P_2	f_2	t_2	c_2	k_2
$(P_1 * P_2)$	$f_{1,2} = \min\{f_1, f_2\}$	$t_{1,2}$	$c_{1,2}$	$k_{1,2}$

A corruption at process p in P_1 can change the evaluation of guards of P_1 and P_2 only at p and its neighbors. This is because the guards of each process involve the local variables at the process itself and its neighbors. So, it is possible that $cons_s(P_2)$ is evaluated *false* at some process $s \in N_p$. If s executes P_2, the effect of the corruption at p spreads in P_2. To prevent this, it is necessary that each process in N_p does not execute P_2 until the variables at p takes the values that they take in the legitimate configuration. By forcing all processes in N_p to stop the

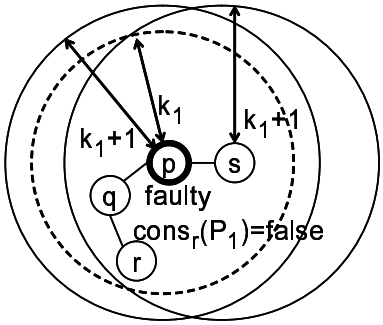


Figure 4.1: Inconsistency range around a faulty process

execution of P_2 during the recovery of P_1, we can prevent the effect of the fault from spreading in P_2. From Assumption 6, there exists at least one process r in $N_p^{k_1}$ for p and in $N_s^{k_1+1}$ for s such that $cons_r(P_1)$ is evaluated *false* during the recovery (See Figure 4.1). *RWFC* strategy is a strategy for fault-containing composition: P_2 should wait the recovery of P_1. To implement *RWFC* strategy, the proposed fault-containing composition stops P_2 by using the contamination radius k_1 of P_1.

We can allow *faulty* processes to execute P_2 before P_1 reaches the legitimate configuration because in an f-faulty configuration, even if faulty processes executes P_2 before P_1 recovers, the number of faulty processes in P_2 is still no larger than f ($\leq f_{1,2}$). What is important is that no *correct* process executes P_2 before P_1 recovers. If some correct process executes P_2 before the recovery of P_1, the number of faulty processes in P_2 may exceed f_2.

The idea of our composition is to make each process p check the inconsistency of each $q \in N_p^{k_1+1}$. For simplicity, we first assume that each process can evaluate $cons_q(P_1)$ for each $q \in N_p^{k_1+1}$ with the *inconsistency detector*. The inconsistency detector guarantees that starting from any f-faulty configuration ($f \leq f_{1,2}$), it provides $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to p in $O(|N_p^{k_1+1}|)$ rounds. We just define the specification and the interface of the inconsistency detector in Section 4.2.1, because our focus is not on the implementation of the inconsistency detector but on the fault-containing composition. We show an implementation of the inconsistency detector in Section 4.3.

4.2.1 Specification of the Inconsistency Detector

The inconsistency detector provides the evaluation of $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to each process $p \in V$. Each process p has two variables, req_p and res_p: when p requests the inconsistency detector to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$, p sets $req_p = 1$, otherwise 0. The inconsistency detector stores the result in res_p that takes a value in $\{true, false, \perp\}$ and p receives the result by reading res_p.

(Note that p cannot change the value of res_p.)

Specification 4 ***The Inconsistency Detector***

(i) *In a legitimate configuration,* $req_p = 0 \wedge res_p = \perp$ *holds at each process* $p \in V$.

(ii) *If process* $p \in V$ *changes* req_p *from* 0 *to* 1 *when* $res_p = \perp$, res_p *takes true or false in* α *rounds with changing the state of only the processes in* N_p^{β}*:*

- *if* $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1) = false$ *holds when the inconsistency detector changes* res_p *from* $\perp$, res_p *takes false.*
- *if* $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1) = true$ *holds when the inconsistency detector changes* res_p *from* $\perp$, res_p *takes true or false. Even when* $res_p = false$ *holds, the inconsistency detector returns* $res_p = true$ *in a constant number of requests.*

(iii) α *and* β *are bounded by some polynomial in* k_1.

(iv) *When* $req_p = 0 \wedge res_p \neq \perp$ *holds at process* $p \in V$, *the inconsistency detector sets* $res_p = \perp$ *in* $O(1)$ *rounds.*

After p requests the evaluation to the inconsistency detector, if $req_p = 1 \wedge res_p = true$ holds, process p can determine that $cons_q(P_1) = true$ holds at each $q \in N_p^{k_1+1}$.

4.2.2 Composition Protocol *RWFC-IcD*

The composition protocol *RWFC-IcD* checks the consistency of P_1 by using the inconsistency detector whenever the upper protocol needs to be executed.

Protocol 4.2.1 shows *RWFC-IcD* for $(P_1 * P_2)$ at process p that provides $f_{1,2}$-fault-containing protocol. For each $i \in \{1, 2\}$, $G(P_i)$ is the disjunction of all guards of protocol P_i at p, and $A(P_i)$ indicates the corresponding action of one of the enabled guards of $G(P_i)$.

Protocol 4.2.1 *RWFC-IcD* for $(P_1 * P_2)$

Actions for process p

S_1 $G(P_1) \longrightarrow A(P_1)$

S_2 $G(P_2) \wedge req_p = 0 \wedge res_p = \perp \longrightarrow req_p = 1$

S_3 $G(P_2) \wedge req_p = 1 \wedge res_p = false \longrightarrow req_p = 0$

S_4 $G(P_2) \wedge req_p = 1 \wedge res_p = true \longrightarrow A(P_2); req_p = 0$

Starting from an f-faulty configuration ($f \leq f_{1,2}$), process p can execute P_1 whenever it has an enabled guard of P_1 by executing S_1. However, when p has an enabled guard of P_2,

p should check the inconsistency of P_1 among $N_p^{k_1+1}$. Process p requests the evaluation to the inconsistency detector by executing S_2 and checks the result with S_3 and S_4. If there is no process $q \in N_p^{k_1+1}$ that finds inconsistency in P_1, then p executes P_2 by executing S_4. Otherwise, p waits the recovery of P_1 by executing S_3.

4.2.3 Correctness Proof of *RWFC-IcD*

First, we show the stabilization of *RWFC-IcD*. Starting from an arbitrary initial configuration, each process can execute P_1 whenever it has an enabled guard of P_1. Thus, it is obvious that eventually P_1 reaches the legitimate configuration and the output of P_1 (the input to P_2) eventually becomes unchanged. After that, if process p requests the inconsistency detector to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$, p always receives $res_p = true$. Thus, the execution of $(P_1 * P_2)$ is that of P_2. So, it is obvious that $(P_1 * P_2)$ eventually reaches the legitimate configuration. The following lemma holds clearly.

Lemma 12 *Starting from an arbitrary initial configuration, RWFC-IcD for* $(P_1 * P_2)$ *eventually reaches the legitimate configuration.*

Secondly, we show the fault-containment of *RWFC-IcD*. Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration in its recovery time and with its contamination number (Lemma 13). Until P_1 reaches the legitimate configuration, each correct process that is a neighbor of a faulty process cannot execute P_2 (Lemma 13). However, a faulty process may execute P_2 before P_1 reaches the legitimate configuration, e.g. if $req_p = 1 \wedge res_p = true$ holds at a faulty process p in an f-faulty configuration ($f \leq f_{1,2}$), then p can execute P_2. Though p executes P_2 before P_1 recovers, the number of faulty processes in the resulting configuration of P_2 is still no larger than f_2. Thus, after P_1 reaches the legitimate configuration, P_2 can reach the legitimate configuration with its fault-containment property.

The composite protocol $(P_1 * P_2)$ via *RWFC-IcD* preserves the fault-containment property of the source protocols (Theorem 4). The performance of the obtained protocol depends on those of P_1, P_2, and the inconsistency detector.

Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 first reaches the legitimate configuration with its fault-containment property.

Lemma 13 *Starting from any f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration with its recovery time and contamination number. During the recovery of P_1, each correct process that is a neighbor of a faulty process cannot execute P_2.*

Proof. Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration with its recovery time and contamination number because S_1 is P_1 itself.

In an f-faulty configuration, $req_q = 0 \wedge res_q = \perp$ holds at each correct process q. If a correct process q is a neighbor of a faulty process and q has an enabled guard in P_2, q changes req_q from 0 to 1 by executing S_2 and the inconsistency detector returns the evaluation of $\bigwedge_{r \in N_q^{k_1+1}} cons_r(P_1)$. From Assumption 6, if P_1 is not in the legitimate configuration, q receives $res_q = false$. So, correct processes neighboring some faulty process(es) do not execute P_2 with incorrect output from P_1. □

Lemma 14 *After P_1 reaches the legitimate configuration, P_2 reaches the legitimate configuration with the recovery time of $t_2\alpha$ and the contamination number of $c_2\Delta^\beta$, where Δ is the maximum degree in G.*

Proof. From Lemma 13, there exist at most f ($\leq f_{1,2}$) faulty processes in P_2 when P_1 reaches the legitimate configuration. Thus, P_2 reaches the legitimate configuration with its fault-containment property: for the variables of P_2, the recovery time and the contamination number is still t_2 and c_2.

However, each process p should check the consistency of P_1 with the inconsistency detector whenever p has an enabled guard of P_2. From Specification 4, this forces each $q \in N_p^\beta$ to change their states and imposes α rounds for p to obtain the result. Thus, in *RWFC-IcD*, it takes $t_2\alpha$ rounds for P_2 to reach the legitimate configuration with the number of $c_2\Delta^\beta$ processes changing their local states. □

From Specification 4, α and β are bounded by some polynomial in k_1.

Theorem 4 *RWFC-IcD provides an $\min\{f_1, f_2\}$-fault-containing protocol $(P_1 * P_2)$ for f_1-fault-containing protocol P_1 and f_2-fault-containing protocol P_2. The recovery time is $(t_1 + t_2\alpha)$ and the contamination number is $\max\{c_1, c_2\Delta^\beta\}$.*

Proof. From Lemma 13 and 14, *RWFC-IcD* executes P_1 and P_2 in the coordinated order and each protocol executes its own recovery actions. So the maximum number of faults that the obtained protocol guarantees fault-containment is $f_{1,2} = \min\{f_1, f_2\}$. From Lemma 14, the recovery time is $(t_1 + t_2\alpha)$ and the contamination number is $\max\{c_1, c_2\Delta^\beta\}$. □

4.3 Inconsistency Detector

In this section, we show an implementation of the inconsistency detector.

The inconsistency detector should provide the communication between process p and each $q \in N_p^{k_1+1}$ to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ whenever p changes req_p from 0 to 1. In the locally shared memory model, process p can read only the local variables at its direct neighbors. Thus,

it is necessary to broadcast the request to each process $q \in N_p^{k_1+1}$ and each $q \in N_p^{k_1+1}$ should return the evaluation of $cons_q(P_1)$ to p.

Recall that the legitimate predicate $L(P_1) \equiv \forall p \in V : cons_p(P_1)$ is a stable predicate on configurations in P_1. Thus, starting from a target faulty configuration, once $L(P_1) = true$ holds, $L(P_1)$ remains *true* thereafter. However, the fault-containment property guarantees that only the processes in the inconsistency range of each faulty process p change their states during the recovery. So, starting from a target faulty configuration, once $cons_q(P_1)$ holds for each $q \in N_p^{k_1+1}$ for a faulty process p, $cons_q(P_1)$ remains *true* at all $q \in N_p^{k_1+1}$ thereafter. Consequently, the inconsistency detector should answer whether there is a configuration where $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ holds between the time when p requests by changing req_p from 0 to 1 and the time the inconsistency detector answers to p by changing res_p from $\perp$ to a value in $\{true, false\}$.

One simple solution for evaluating a stable predicate is to use *PIF (Propagation of Information with Feedback)* protocols that take a snapshot of global configurations by broadcasting a request to all processes and gathering feedbacks from all processes.

However, we do not need any global detection but the local detection among $N_p^{k_1+1}$ for process p. One way to involve $N_p^{k_1+1}$ into some task is to use the breadth first tree of height $(k_1 + 1)$ rooted at p. Whenever process p changes req_p to 1, p constructs the breadth first tree and by using a PIF protocol on the breadth first tree, p broadcasts the request to each $q \in N_p^{k_1+1}$ and q feedbacks the evaluation of $cons_q(P_1)$ to p. We can use the breadth first tree construction protocol in [30] by setting the height of the tree $k_1 + 1$.

However, this simple implementation cannot provide the correct evaluation of the predicate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to p. Because each process executes P_1 during the request and feedback of a PIF protocol, the evaluation of $cons_q(P_1)$ at $q \in N_p^{k_1+1}$ may change during the feedback: e.g. after q sends $cons_q(P_1) = true$ as a feedback, if the evaluation of $cons_q(P_1)$ changes from *true* to *false* (it may be caused by some state change of the processes in N_q), p cannot obtain the correct evaluation of $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$.

Generally, to evaluate a stable predicate among processes, PIF is used twice. The first PIF propagates the request to each process and each process starts to observe the stable predicate. The second PIF gathers the result of observation at each process via the feedback of PIF. In this way, one can evaluate a stable predicate on configurations.

Cournier et al. proposed a snap-stabilizing PIF protocol for arbitrary networks [12]. Their protocol *PIF* guarantees that each process returns the feedback after all processes in V received the request. Thus, by using *PIF*, we can collect the observation of the stable predicate with a single PIF execution.

We allow each process p to execute *PIF* independently in parallel so that each process

$q \in N_p^{k_1+1}$ can evaluate $\bigwedge_{r \in N_q^{k_1+1}} cons_r(P_1)$ when q changes req_q from 0 to 1. This is done, for example, by attaching the ID of q to the broadcast and feedback. This imposes additional memory of size of $O(|N_p^{k_1+1}| \log n)$ at p to manage different trees while this does not impose additional time complexity.

We modify *PIF* as follows:

(i) process p constructs the breadth first tree of height $(k_1 + 1)$ rooted at p when it changes req_p from 0 to 1.

(ii) process q starts to observe $cons_q(P_1)$ when it receives the request of the PIF protocol. If $cons_p(P_1) = true$ holds during the observation, p records it.

(iii) q returns the result of the observation to p with the feedback of *PIF*.

The snap-stabilization property of *PIF* guarantees that starting from an arbitrary initial configuration, whenever the root process begins the broadcast, every process receives the broadcast and the root process receives feedback from every process in $O(N)$ rounds. Thus, in our implementation, the broadcast and feedback take $O(N_p^{k_1+1})$ rounds. The breadth-first tree is constructed in $O(k_1 + 1)$ rounds. Thus, p receives the feedback from all processes in $N_p^{k_1+1}$ in $O(|N_p^{k_1+1}|)$ rounds. Consequently, the value of α in Specification 4 is a polynomial in k_1. Because only the processes in $N_p^{k_1+1}$ change their states, the value of β in Specification 4 is $k_1 + 1$. So, the condition (iii) of Specification 4 is satisfied.

To satisfy the condition (iv) of Specification 4, the inconsistency detector should check req_p and res_p, and whenever $req_p = 0 \wedge res_p \neq \perp$ holds at p, it should change res_p to $\perp$.

4.4 Concluding Remarks

In this chapter, we proposed a fault-containing composition technique that utilizes spatial containment property of source protocols. Because fault-containing protocols provide the temporal containment property and/or the spatial containment property, *RWFC-IcD* is the complement of *RWFC-LNS*.

We utilize the spatial containment property of fault-containing protocols to check the inconsistency of the lower protocol. Then, we defined and implemented the inconsistency detector that enables each process to communicate with the processes in its inconsistency range of the lower protocol. Our implementation is based on an existing snap-stabilizing PIF protocol and we modified *PIF* [12] so that it is executed among the processes in the inconsistency range of the lower protocol. The performance of obtained protocol depends on the inconsistency detector. Though the PIF protocol imposes additional communication overhead and execution time,

the effect is contained around faulty processes in the inconsistency range of the lower protocol. The inconsistency range of the lower protocol is limited because the lower protocol is fault-containing. Thus, the overhead imposed by the PIF protocol is small and do not spread over the entire network.

To accelerate the composite protocol by *RWFC*, we can use the legitimacy of only output variables of the lower protocol (called *output legitimacy*) instead of legitimacy of input, inner, and output variables. This is because the upper protocol just uses the output variables of the lower protocol as its input. However, to adopt output legitimacy, it is necessary that when the system starts from a target faulty configuration, once the lower protocol reaches the output legitimate configuration, it does not change the values of output variables thereafter. Note that not all the fault-containing protocols provide this property for output legitimacy.

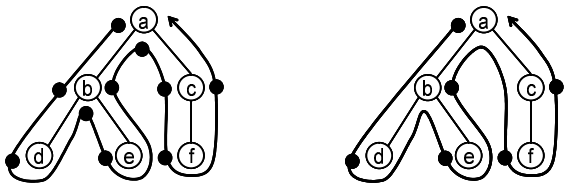
Chapter 5

Ring Embedding Preserving Fault-containment Property

In this chapter, we present a ring embedding on an arbitrary rooted tree that enables simulation of fault-containing ring protocols on an arbitrary rooted tree.

The most desirable ring embedding is the one along a Hamiltonian cycle of the tree network, but finding a Hamiltonian cycle is computationally intractable. To embed a ring on an arbitrary network, one way is to embed the ring in a spanning tree of the network. Commonly used ring embeddings on a tree are the Euler tour [14] (Figure 5.1(a)), the one proposed by Sekanina [37, 49] (Figure 5.1(b)), and an embedding similar to Sekanina's, proposed by Arora et al. [2]. We observe that adjacent processes in the tree remain adjacent in the Euler tour, but not in Sekanina's or Arora's ring embeddings; Sekanina's and Arora's embeddings have the *dilation* (the maximum distance in the tree between consecutive processes in the ring) of three. Also, each process in the tree corresponds to a single process in the ring in Sekanina's and Arora's embeddings, while in the Euler tour a process in the tree is duplicated a number of times equal to its degree. Thus for a tree with n processes, the length of the ring based on the Euler tour is $2n - 2$, respectively n for Sekanina's and Arora's ring embeddings.

Simulation of protocols designed for simple networks (e.g. rings) on an arbitrary network facilitates the design of new protocols. However, it involves the difficulty described below when applied to fault-containing protocols. Euler tour embedding cannot preserve the fault containment property of ring protocols. This impossibility is due to the fact that a tree process appears in the Euler ring several times: a faulty process in the tree is treated as multiple faults in the ring. The number of faults in the ring may exceed the maximum number of faults that the fault-containing ring protocol can tolerate. Sekanina's embedding is the first step in preserving fault-containment of a ring protocol in a tree. The one-to-one node embedding ensures that a



(a) Euler tour ring (b) Sekanina's ring

Figure 5.1: Ring embedding on a tree

fault that affects a single process in the tree can be treated as a single faulty process in the ring. However this embedding alone cannot ensure fault-containment completely since the links of the ring go through some intermediate process(es) that can be corrupted by a fault, thus corrupting the ring communication.

The proposed simulation protocol *RET* (ring embedding protocol on an arbitrary tree) enables fault-containing protocols to be executed on Sekanina's ring embedded on an arbitrary rooted tree. The dilation of Sekanina's embedding is three, thus neighboring processes in the virtual ring are not necessarily neighboring processes in the tree. Additionally, any process p, of degree δ_p, is an intermediate process for $\delta_p - 1$ embedded ring links. Thus a single fault at process p can affect communication for δ_p virtual links of Sekanina's embedding that pass through p.

To overcome this difficulty, we force eahc sender process to send each data five timers and the destination process to compute majority of the received data to exclude the corrupted data. We use a communication synchronizer [45] among neighboring processes so that for each data to be routed at most two corrupted pieces of the data are read at the endpoint process, and We force each piece of data to be relayed five times and the endpoint process to apply the majority computation on them. Because there are at most two corrupted pieces of data in each set of five pieces of data, the corrupted data is removed at the endpoint process. Repeating the communication five times before delivering the data causes a slowdown in executing the ring protocol in the tree, but overall the slowdown of the ring protocol is proportional to the maximum degree of the tree.

Related works. We introduce the notion of *causal simulation*, a method for applying ring protocols to trees that preserves the fault-containment property. Lynch defined the *simulation* relation between two different protocols that requires one protocol traces every global configuration of the other protocol [41]. However, our simulation protocol cannot provide simulation relation between the execution of the original fault-containing ring protocol and the execution of the simulation protocol. This is because each virtual link have different communication delays. Causal simulation preserves the read/write causality of each data of the execution of original

protocol and this is strong enough to guarantee that the simulating protocol executes the same task as the original protocol. This is because most of reactive and non-reactive tasks are based on read/write causality.

This chapter is organized as follows. In Section 5.1 we present system models and the definition of the vertex bijective ring. In Section 5.2 we define the causal simulation. In Section 5.3 we present protocol *RET* that provides a causal simulation of ring protocols by simulating the communication link on the *vertex bijective ring*. In Section 5.4 we show how protocol RET can be used to design a 1-fault-containing leader election protocol in arbitrary trees. We conclude this chapter in Section 5.5.

5.1 Preliminary

In this chapter, we consider self-stabilizing protocols and fault-containing protocols for reactive and non-reactive problems. Hence, the set of legitimate configurations of a problem is defined by Definition 2.

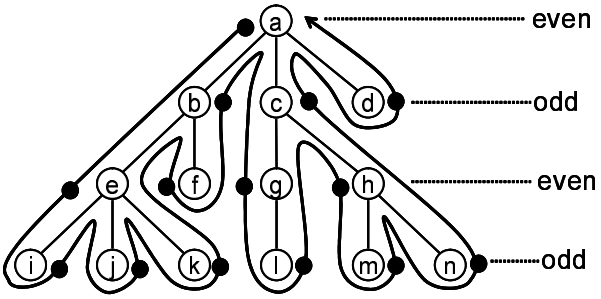
In this chapter, when a process is corrupted by a fault we consider that the process has executed a *faulty action* and the process is called *faulty*.

We embed a *vertex bijective ring* on a rooted tree and simulate a fault-containing ring protocol on the embedded ring.

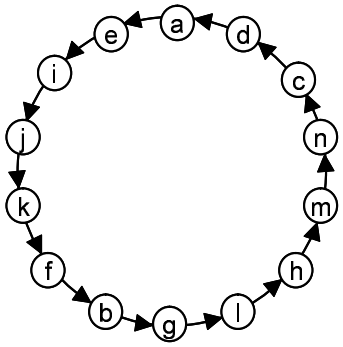
Definition 7 ***Vertex Bijective Ring***
Given a tree $T = (V, E)$*, a* vertex bijective ring *of* T *is any ring* $R = (V, E')$ *embedded on* T *such that each process of tree* T *appears only once on the ring* R*.*

The processes and the links of a *vertex bijective ring* R are called *virtual*, and the processes and the links of T are called *real*. The *dilation* of R in T is the maximum distance in T between any neighboring processes in R.

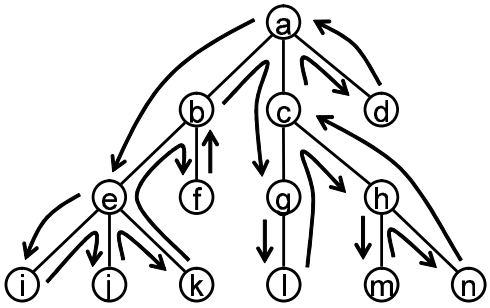
Sekanina's ring embedding [49] is an example of a vertex bijective ring of a tree with dilation of three. It can be described as a *preorder-postorder traversal* of the tree. Given a tree $T = (V, E)$ rooted at process r, the processes in T are divided into even and odd-level processes such that: (i) the root is at even level, and (ii) a process is at odd (even) level iff its parent is at even (odd) level. The *preorder-postorder traversal* starts with the root process r, then continues along a depth-first traversal. A process $p \in V$ is deployed on the virtual ring as follows. If p's level is even, p is deployed as preorder traversal, that is, p is deployed on the ring before all its descendants in the tree. If p's level is odd, p is deployed as postorder traversal, that is, p is deployed on the ring after all its descendants.



(*a*) *preorder-postorder traversal*



(*b*) The Virtual Ring



(*c*) Routing in the Virtual Ring

Figure 5.2: Preorder-postorder traversal

Two adjacent processes in a *preorder-postorder traversal* are connected by a *virtual link*. To form a ring, a virtual link is added between the last visited process and the root process r. The *preorder-postorder traversal* for a given tree topology is shown in Figure 5.2(a). The obtained ring is shown in Figure 5.2(b). Process a (the root of the tree T) is deployed first. The successor of a in the traversal is e because b's level is odd and the next preorder process is e. Process e, whose level is even, is deployed for the ring before e's descendants (processes i, j, and k) are deployed. Process b, whose level is odd, is selected for the ring after its descendants (processes e, i, j, k, and f) are deployed.

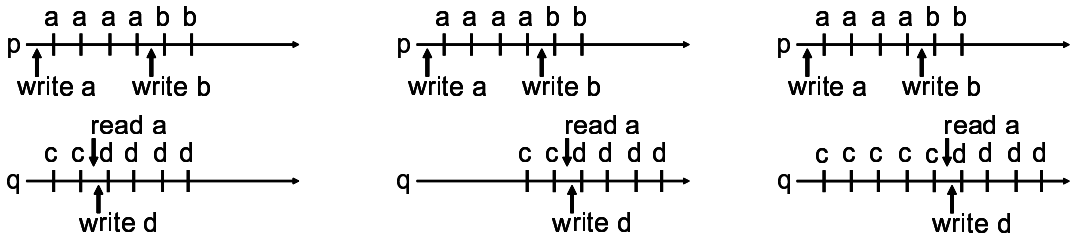
Figure 5.2(c) shows the virtual links of the embedded ring presented in Figure 5.2(b). Each virtual link is a path of at most three real (tree) links (e.g., a virtual link (b, g) is a path (b, a, c, g) in the tree).

5.2 Causal Simulation

In this section we give a formal definition of *causal simulation*. We first introduce the causal simulation of a fault-free and non-stabilizing case, then we progress into a self-stabilizing case and a fault-containing case. The idea of causal simulation of protocol P_v designed for topology T_v is that a protocol P_r designed for topology T_r executes the same task as P_v.

We now define what it means that one protocol P_r defined for topology T_r provides a causal simulation of another protocol P_v defined for topology T_v, both using locally shared memory model. Besides the variables of P_v, a process in P_r may have another set of variables. The variables that are common to P_v and P_r are called *common* and the rest are called *non-common*. Thus the projection of a process state in P_r on to its common variables represents its state in P_v. Given an execution E_r of P_r, this projection defines the behavior in P_v. $\mathcal{R}(E_r)$ represents the execution in P_v obtained by the projection of E_r and removing some stuttering configurations in it. Causal simulation guarantees that any execution E_r of P_r has a corresponding execution E_v of P_v such that we can obtain $\mathcal{R}(E_r)$ by a shift operation on time-space diagram of E_v. We call this shift operation as *causal shift* since the operation preserves the read/write causality of the original execution in a sense that any data read was written before.

Generally, when executing a self-stabilizing protocol or a fault-containing protocol, a process may write the same value on one register repeatedly. For example, let a process write value a to one register three times. We consider this as write actions of three different data: e.g. $a_{(1)}$, $a_{(2)}$, $a_{(3)}$ are written in the register successively. Causal shift should preserve the read/write



(a) Time-space diagram of P_v (b) State shift (i) of (a) (c) State addition (ii) of (a)

Figure 5.3: An example of causal shift

causality for each $a_{(1)}$, $a_{(2)}$, $a_{(3)}$.

Definition 8 ***Causal Shift***

Given an infinite execution E_v of P_v, a causal shift *modifies the time-space diagram of E_v as follows:*

(i) shift states on the temporal lines of processes, but keep each write event of some data precedent to any read event of that data, and

(ii) add some copies of the initial state of a process as the first states of the process so that the initial states of all processes are aligned.

The time-space diagram obtained from execution E_v by causal shift represents a sequence of configurations denoted by $E_{v|cs}$. $E_{v|cs}$ is not uniquely defined by E_v.

Figure 5.3 shows an example of a causal shift where we focus on the read/write causality of data a that is written to the register at process p and after that is read by one of p's neighbors, q. Process q changes its state from c to d according to data a (Figure 5.3(a)). Because causal shift should preserve that the data read should always be written before, we can shift the states at q to right. Let the operation (i) of causal shift shift the temporal line of q as Figure 5.3(b). Then, the first state c of q is copied and the initial states of p and q are aligned. The sequence of configuration obtained from P_v is given in Figure 5.3(c). Causal shift does not violate the read/write causality of the original execution.

We say two states s_p of process p and s_q of process q ($p \neq q$) are *concurrent* in an execution iff s_p and s_q have no relation in the sense of read/write causality. From Definition 8 the following remark follows immediately.

Remark 3 *For any configuration σ in $E_{v|cs}$, all local states that are part of σ are concurrent states in E_v.*

Any configuration sequence obtained by a causal shift from an execution denotes another execution in an asynchronous message passing system. However, in locally shared memory model,

$E_{v|cs}$ does not necessarily denote an execution. For example, in Figure 5.3(c) process q cannot read a when the state of p is b. Causal simulation is weaker than the *simulation relation* defined by Lynch [41]. Lynch defined a simulation relation between two protocols such that for any execution of one protocol, every computation step is traced by the other protocol. This means every global configuration of the original protocol appears in the simulation protocol. Causal simulation does not trace global configurations of original executions but preserves the local behavior of each process and the read/write causality. This is sufficient for many problems such that the legitimacy of the problem is defined by the read/write causality: e.g. leader election problem, spanning tree construction problem, token circulation problem, and so on.

Definition 9 ***Causal Simulation of Fault-free Non-stabilizing Protocols***
A protocol P_r defined for a topology $\mathcal{T}_r$ provides a causal simulation of another protocol P_v defined for a topology $\mathcal{T}_v$ (with the same process set as $\mathcal{T}_r$) on a locally shared memory model iff for any infinite execution E_r of P_r, there exists an infinite execution E_v of P_v such that we obtain $\mathcal{R}(E_r)$ from $E_{v|cs}$.

Starting from a predefined good initial configuration, it may be possible to provide a causal simulation of the original protocol from the initial configuration. However, self-stabilizing protocols start from an arbitrary initial configuration. We relax the restriction for a causal simulation of self-stabilizing protocols: starting from an arbitrary initial configuration, P_r eventually provides the causal simulation of P_v. Thus, to preserve the self-stabilization of P_v, causal simulation guarantees that any execution of P_r starting from an arbitrary initial configuration has a suffix whose projection on to the common variables is obtained from some causal shift of some execution of P_v. We denote an infinite suffix of an infinite execution E by *suff*(E).

Definition 10 ***Causal Simulation of Fault-free Self-stabilizing Protocols***
A protocol P_r defined for a topology $\mathcal{T}_r$ provides a causal simulation of another self-stabilizing protocol P_v defined for a topology $\mathcal{T}_v$ (with the same process set as $\mathcal{T}_r$) on a locally shared memory model iff for any infinite execution E_r of P_r starting from any arbitrary initial configuration, there exists an infinite execution E_v of P_v such that we obtain suff($\mathcal{R}(E_r)$) from $E_{v|cs}$.

A configuration σ_r of P_r is *cs-legitimate* iff for any execution starting from σ_r the projection of it is obtained by a causal shift of some execution of P_v starting from a legitimate configuration of P_v. Since P_r do the same task as P_v and P_v is a self-stabilizing protocol, P_r eventually reaches a cs-legitimate configuration.

Remark 4 *Starting from any arbitrary initial configuration, P_r eventually reaches a cs-legitimate configuration.*

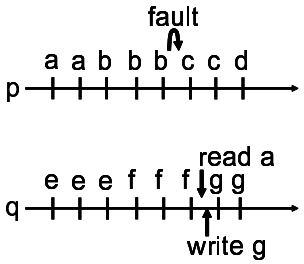


Figure 5.4: Faults in P_r

To preserve the fault containment of P_v, intuitively the following conditions should be satisfied: if E_r starts with a cs-legitimate configuration σ_r of P_r and the fault corrupts some processes in the first computation step, $\mathcal{R}(E_r)$ should be a suffix of a causal shift of some execution E_v of P_v such that all the faulty processes experience faulty actions in E_v in a legitimate configuration σ_v at the same time in E_v and the faulty actions change all the common variables in the same way as E_r. If the number of faulty processes is not larger than the maximum number for which E_v guarantees fault containment, E_r shows the fault-containing recovery because E_r provides a causal simulation of E_v.

However, it is not guaranteed that the above condition holds. This is because E_r is obtained by a causal shift of some execution E_v of P_v. A causal shift shifts states on the temporal line of processes of E_v and may also shift the faulty actions in E_v executed at the same time at different processes. Furthermore, a causal shift can produce a configuration of E_r that may never appear in E_v. For example, based on the causal shift, such execution of P_r can exist (Figure 5.4). After the fault corrupts some process p, a non-faulty process q reads the data that p holds before the corruption. Thus, the projection of global configuration just before the corruption cannot appear in P_v.

Still, it is guaranteed that there is some execution E_v of P_v such that all faulty processes execute faulty actions in a legitimate configuration of P_v and the projection of the states before and after the corruption at faulty processes is same as those in E_v.

Remark 5 *There exists an execution of P_v such that all faulty processes execute faulty actions in a legitimate configuration of P_v and those faulty actions change all the common variables in the same way as E_r.*

Proof. Consider the case that faults corrupted two processes in a cs-legitimate configuration σ_r of P_r. These two faults occur at processes p and q, and p (respectively, q) changes its state from s_p to s'_p (from s_q to s'_q, respectively). Let $\mathcal{R}(s_p)$ represent the projection of s_p on to the common variables.

Since σ_r is a cs-legitimate configuration of P_r, the projection of σ_r is obtained by causal shift of some execution of P_r starting from a legitimate configuration.

Let us consider fault-free executions first. From some fault-free execution, $E_r^* = \sigma_r, \sigma_{r+1}, \ldots$, E_r is obtained by corrupting some processes after σ_r. Such E_r^* is obtained from some execution of P_v starting from a legitimate configuration since σ_r is a cs-legitimate configuration. Let $\Pi(\sigma_r)$ be a set of executions of P_v from which we can obtain the projection of any execution of P_r starting from σ_r by $\mathit{suff}(E'_{v|cs})$ where $E'_v \in \Pi(\sigma_r)$ and E'_v starts from a legitimate configuration of P_v.

For any $E'_v \in \Pi(\sigma_r)$, p takes the state $\mathcal{R}(s_p)$ in some configuration σ_p and q takes the state $\mathcal{R}(s_q)$ in some configuration σ_q. For contradiction, assume that σ_p and σ_q do not coincide in any execution $E'_v \in \Pi(\sigma_r)$. If $\mathcal{R}(s_p)$ and $\mathcal{R}(s_q)$ are concurrent states, then there exists at least one execution in $\Pi(\sigma_r)$ such that these two states coincide in one configuration. However, if there is no such execution in $\Pi(\sigma_r)$, then there exists some read/write causality between $\mathcal{R}(s_p)$ and $\mathcal{R}(s_q)$. Thus, $\mathcal{R}(s_p)$ and $\mathcal{R}(s_q)$ are not concurrent states in P_v and this conflicts with Remark 3.

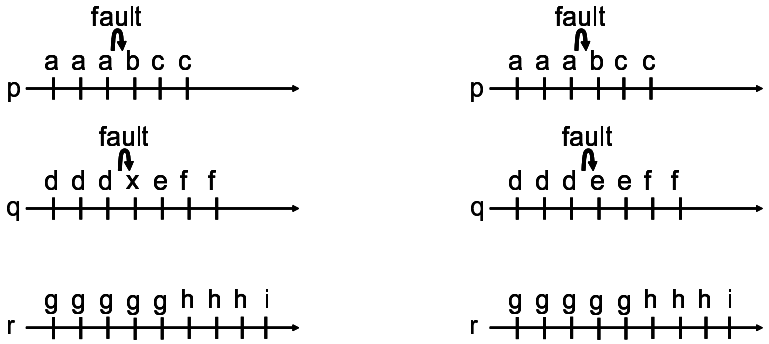
Thus, there exists some execution $E_v^* \in \Pi(\sigma_r)$ such that $\mathcal{R}(s_p)$ and $\mathcal{R}(s_q)$ coincide in a configuration σ_v. Because σ_r is a cs-legitimate configuration of P_r, σ_v is a legitimate configuration of P_v.

We considered that the number of faulty processes is two, but when the number of faulty processes is bigger than two, we can conclude in the same way. □

In E_r, the fault corrupts some processes and changes the configuration from σ_r to σ'_r. This corresponds to that in E_v^*, the fault corrupts some processes and changes the configuration from σ_v to σ'_v. If the number of faulty processes is not larger than the maximum number of faulty processes that P_v guarantees fault-containment, the execution of P_v after the corruption shows a recovery of fault-containment and E_r also shows the recovery of fault-containment.

When a fault corrupts some processes, the corrupted process cannot provide causal simulation immediately after the corruption. For example, common variables at a faulty process may change after the corruption because non-common variables were also corrupted by the fault. In this case, common variables may flutter and the state of that process in the projection may also flutter. However, we can ignore this repetition of fluttering states if the repetition is finite and the fluttering states do not affect other processes in the projection: any neighbor in P_v does not change its state according to the fluttering states at a faulty process.

Let $\mathcal{R}'(E_r)$ be obtained from $\mathcal{R}(E_r)$ by replacing some local states that appear consecutively and immediately after each faulty action with the state that follows the last replaced state. $\mathcal{R}'(E_r)$ is not uniquely defined by $\mathcal{R}(E_r)$. Figure 5.5 shows an example of $\mathcal{R}'(E_r)$ where the state x at process q is replaced with the following state e.



(*a*) $\mathcal{R}(E_r)$ for execution E_r of P_r (*b*) $\mathcal{R}'(E_r)$

Figure 5.5: An example of $\mathcal{R}'(E_r)$

In $\mathcal{R}'(E_r)$, the replaced states do not affect other processes in the projection while the replacing state may affect other processes in the projection. $\mathcal{R}'(E_r)$ enables the simulating protocol P_r to have other variables than common variables since it may take some steps (or rounds) for P_r to recover both common and non-common variables after the corruption. Clearly, Remark 5 holds when we use $\mathcal{R}'(E_r)$ instead of $\mathcal{R}(E_r)$.

Definition 11 ***Causal Simulation of Fault-containing Protocols***
A protocol P_r defined for a topology $\mathcal{T}_r$ provides a causal simulation of another fault-containing protocol P_v defined for a topology $\mathcal{T}_v$ (with the same process set as $\mathcal{T}_r$) on a locally shared memory model iff for any execution E_r of P_r starting from any cs-legitimate configuration where all faulty actions occur in the first step and all the other actions are correct ones, there exists an execution E_v of P_v such that E_v starts from a legitimate configuration of P_v and a fault corrupts the same processes at a time and there exists $\mathcal{R}'(E_r)$ that we obtain from suff$(E_{v|cs})$.

If P_r satisfies both Definitions 10 and 11 for fault-containing self-stabilizing protocol P_v, P_r provides the causal simulation of P_v preserving the self-stabilization property and fault-containment property of P_v.

Causal simulation preserves the behavior of each process and the read/write causality of original execution. When a fault corrupts some processes in a cs-legitimate configuration of P_r, causal simulation guarantees there exists some execution of P_v that experiences the corruption of the same process set at a time. Consequently, for a fault-containing self-stabilizing protocols P_v, causal simulation guarantees the fault containment property if the number of corrupted processes is smaller than or equal to the number for which P_v guarantees fault containment.

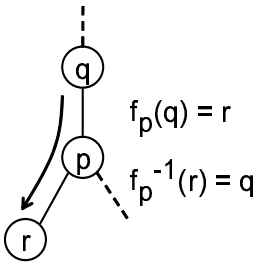


Figure 5.6: Local routing function at p

5.3 Causal Simulation Framework

We define protocol *RET* for a tree. We show that it provides a causal simulation of a unidirectional ring protocol on the tree and preserves the fault-containment of the ring protocol (Theorem 5).

We embed a *vertex bijective ring* on a tree. Our local routing function is similar to Arora's paper [2]. Each process p is part of the data in δ_p virtual links and maintains a local routing function $f_p()$. For $q \in N_p \cup \{p\}$, $f_p(q)$ returns some process in N_p or p to which p relays the data from q: $f_p(q) = r$ with $r \in N_p \cup \{p\}$ if p needs to forward to r the data from q. The inverse of $f_p(q)$, $f_p^{-1}(q)$, returns p or some neighbor of p from which p should relay to q: $f_p^{-1}(q) = r$ with $r \in N_p \cup \{p\}$ if p should read from r the data for q. The following relation always holds (see Figure 5.6): $f_p(q) = r \Leftrightarrow f_p^{-1}(r) = q$.

The routing function $f_p()$ at each process p is defined as follows. Let $N_p[0]$ represent p's parent on the tree and $N_p[i]$ $(1 \leq i < \delta_p)$ represent p's children (thus, $N_p[i]$ $(0 \leq i < \delta_p)$ is constant). At the root process r, $N_r[0] = \perp$ and $N_r[\delta_r] \neq \perp$ $(\delta_r \geq 1)$.

1. For the root process r ($N_r[0] = \perp$, $N_r[\delta_r] \neq \perp$),

$$f_r(q) = \begin{cases} N_r[1] & \text{if } q = r \text{ and } r \text{ has a child} \\ & (N_r[1] \text{ is the first child of } r) \\ N_r[0] & \text{if } q = r \text{ and } r \text{ has no children} \\ N_r[i+1] & \text{if } q = N_r[i] \text{ and } 1 \leq i < \delta_r \\ r & \text{if } q = N_r[\delta_r] \end{cases}$$

2. For a non-root, even-level process p,

$$f_p(q) = \begin{cases} N_p[1] & \text{if } q = p \text{ and } p \text{ has a child} \\ & (N_p[1] \text{ if the first child of } p) \\ N_p[0] & \text{if } q = p \text{ and } p \text{ has no children} \\ p & \text{if } q = N_p[0] \\ N_p[i+1] & \text{if } q = N_p[i] \text{ and } 1 \leq i < \delta_p - 1 \\ N_p[0] & \text{if } p \text{ has a child and } q = N_p[\delta_p - 1] \end{cases}$$

3. For an odd-level process p,

$$f_p(q) = \begin{cases} N_p[0] & \text{if } q = p \\ N_p[1] & \text{if } q = N_p[0] \text{ and } p \text{ has a child} \\ & (N_p[1] \text{ is the first child of } p) \\ p & \text{if } q = N_p[0] \text{ and } p \text{ has no children} \\ N_p[i+1] & \text{if } q = N_p[i] \text{ and } 1 \leq i < \delta_p - 1 \\ p & \text{if } p \text{ has a child and } q = N_p[\delta_p - 1] \end{cases}$$

For process c in Figure 5.2(c), $f_c(a) = g$, $f_c(g) = h$, $f_c(h) = c$, $f_c(c) = a$.

A virtual link of *vertex bijective ring* consists of at most three (tree) links and at most two intermediate processes. These intermediate processes relay the data between endpoints in a store-and-forward manner. We consider the case of a virtual link along which the intermediate processes are corrupted but the endpoints are not. If an endpoint is corrupted by the fault, the virtual process of that endpoint is also corrupted. If we allow the corrupted data to be read at the endpoint of the virtual link, the fault may spread in the entire system unhindered.

To prevent unlimited propagation of corrupted data from an intermediate process to an endpoint process of a virtual link, we synchronize the communication of a process with its immediate neighbors. Between two consecutive communications of a process p with its neighbor q, we force p to communicate with all its other neighbors. To this end, we use the mechanism of link alternator [45]. There are also tree synchronizers [32, 7], however these synchronizers are not snap-stabilizing because a faulty process can communicate twice consecutively with the same neighbor before communicating with other neighbors.

The link alternator is snap-stabilizing and ensures synchronization immediately after the fault. In the link alternator protocol, each process p has a pointer $comp_p$ indicating a process in N_p. When two neighboring processes point at each other (e.g. $comp_p = q$ and $comp_q = p$ for some process $q \in N_p$), the two processes can communicate. After that they change the pointer to another neighbor. The ordering of the neighbors is determined by the topology and each process can communicate with its neighbors in a round robin fashion. The link alternator protocol is

snap-stabilizing; every configuration of the protocol is legitimate. Starting from an arbitrary initial configuration, the protocol ensures that each process can communicate with its neighbors in a round robin fashion. Thus, in every execution, between two consecutive communications with the same neighbor, each process communicates with other neighbors.

5.3.1 Causal Simulation Protocol RET

In a locally shared memory model, a process executes the following three steps: (1) *reads* the local variable(s) of the immediate neighbor(s), (2) executes some local computation, (3) *writes* into its own local variable(s). When process p receives five pieces of data from its predecessor in the virtual ring, p computes majority of them. The result is then *delivered* at p that corresponds to *read* action in the virtual ring. Then, p executes the computation of the original protocol and updates the common variables, which corresponds to a *write* action to common variables in the virtual ring.

Protocol *RET* (Protocol 5.3.1) uses the following variables. Each process p has a variable ω_p used by the ring protocol. [1]. Variable c_p denotes the remaining number of times p should read the data from its predecessor in the ring (through possible intermediate processes) and c_p takes a value in the set $\{0, \ldots, 4\}$. In the original ring protocol, process p updates ω_p by executing a generic action called $Action()$. $Action()$ has two parameters: the current content of p, ω_p, and the data relayed from its predecessor in the ring. By default, $Action(\omega_p, \perp) = \omega_p$ (if no data is delivered at p, then do nothing).

For the routed data, process p uses the variables:

1. $w_p^{\dagger}(q)$, called *contents table*, keeps the data sent by $f_p^{-1}(q)$ through p that needs to be relayed to process q, $q \in N_p \cup \{p\}$. If $f_p^{-1}(q) = p$ then $w_p^{\dagger}(q) = \omega_p$.

2. $w_p^{\ddagger}$, called *cache table*, keeps the latest five data relayed from p's predecessor in the ring. Two operations are defined on the table: $append(w_p^{\ddagger}, w)$ appends w to $w_p^{\ddagger}$, and $ext(w_p^{\ddagger})$ returns the five entries.

Function $maj(w_p^{\ddagger})$ returns the majority of $ext(w_p^{\ddagger})$ or $\perp$ if there is no majority. Predicate $comm_p(q)$ is controlled by the communication synchronizer and is *true* when process p can communicate with some neighbor q, $q \in N_p$.

Action C_1 corrects the value of the counter c_p (if outside the range $[0,\ldots,4]$, then it is set to 4), and also ensures that the entry $w_p^{\dagger}(f_p(p))$ of the contents table is equal to ω_p ($f_p(p)$ is the neighbor of p towards p's successor in the ring).

Action A_1 simulates the ring communication on the tree. Condition $f_p(q) \neq p$ implies that

[1] For simplicity, we use one common variable for each process but without loss of generality this can be seen as a composite of multiple common variables.

Protocol 5.3.1 *RET (Ring Embedding protocol on an arbitrary rooted Tree)*

Predicate $OK_c(p) \equiv (0 \leq c_p < 5) \wedge w_p^\dagger(f_p(p)) = \omega_p$

Actions for any process p

C_1 $c_p < 0 \vee c_p \geq 5 \vee w_p^\dagger(f_p(p)) \neq \omega_p \longrightarrow c_p = 4;\ w_p^\dagger(f_p(p)) = \omega_p$

A_1 $OK_c(p) \wedge comm_p(q) \longrightarrow$
// read from q
if $f_p(q) \neq p$ **then** $w_p^\dagger(f_p(q)) = w_q^\dagger(p)$
else
$append(w_p^\ddagger, w_q^\dagger(p))$
if $c_p > 0$ **then** $c_p - -$
else
$\omega_p \leftarrow Action(\omega_p, maj(w_p^\ddagger))$
$w_p^\dagger(f_p(p)) = \omega_p$
$c_p = 4$

the data read from q needs to be relayed further. When the counter c_p is 0 the majority is applied (function maj) to the content of $\omega_p^\ddagger$ relayed from its predecessor in the virtual ring. If the result is not empty, we say that the result is *delivered* at p. Then, p executes $Action()$ and updates ω_p with the result. The counter c_p is reset to 4 and the new content of ω_p is relayed to its successor in the ring.

5.3.2 Correctness Proof of *RET*

For some process p, let q and r be its predecessor and successor in the ring.

In any execution of *RET*, if q changes ω_q to s with $Action(\omega_q, maj(\omega_p^\ddagger))$ and no fault occurs at q during $\omega_q = s$, then p stores at least five s's in $\omega_p^\ddagger$ (Proposition 1). Independent of the value of c_p at the time q changes ω_q to s, if p stores five s's in $\omega_p^\ddagger$, then s is delivered at p (Proposition 2). A configuration σ of *RET* is a legitimate configuration iff in any execution starting from σ Proposition 1 and Proposition 2 hold.

A legitimate configuration is reached in finite time once each process has reset its counter c_p to 4 at least once (Lemma 15). The time complexity depends on the synchronizer used for communication between neighboring processes.

Since dilation is three, a fault at the intermediate processes on a virtual link may corrupt

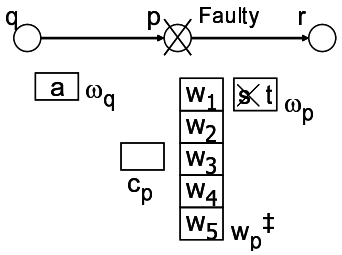


Figure 5.7: Fault at p on the virtual ring

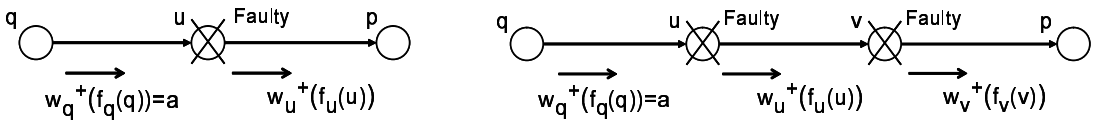
at most two pieces of data out of five, thus by applying a majority function the corrupted data is eliminated. If neither q nor p are faulty and $\omega_q = s$, then either data s is delivered at p or is lost (but no other data is delivered) if some fault occurs at the intermediate processes on the virtual link between q and p (Lemma 16).

The fault at p can corrupt p's variables: ω_p, c_p, $w_p^\dagger$, and $w_p^\ddagger$. Assume that the fault had changed ω_p from a value s to a value $t \neq s$, c_p to some value in the set $\{0, \ldots, 4\}$ (Action C_1 corrects it otherwise), and the top five entries of $w_p^\ddagger$ are $w_1, \ldots, w_5$ (Figure 5.7). The contents table $w_p^\dagger$ is correctable: the entries for all $q \neq f_p(p)$ in the contents table $w_p^\dagger$ will be corrected (Lemma 16). The entry $w_p^\dagger(f_p(p))$ (data that p forwards to its successor in the ring) is equal to ω_p (Action C_1 corrects it otherwise).

In the ring, process p reads the contents of process q that is a value s. This corresponds in the tree that p stores five s's in $\omega_p^\ddagger$ (Proposition 1). Propositions 1 and 2 deal with fault-free cases. We show that a fault at p that occurred during this communication may affect it at most twice: Only data s and another data that p took before s may be lost, but no data before that or after s is lost and the state of p in the virtual ring, ω_p, is corrupted at most once by the fault in the virtual ring (Lemma 17). We can then conclude that a fault may cause a loss of at most three pieces of data per virtual link (Lemma 18). Lemmas 16, 17, and 18 cover all possible timings of faults for a virtual link. Then, we will show the data loss is acceptable in the causal simulation of ring protocols (Theorem 5).

Proposition 1 *If process q changes its content ω_q to s by Action$(\omega_q, maj(\omega_p^\ddagger))$, and p is the successor of q in the virtual ring, then p stores at least five s's in $\omega_p^\ddagger$.*

Proof. Whenever the contents of q (stored in ω_q and $w_q^\dagger(f_q(q))$) is changed, the counter c_q is reset to 4. Whenever the current value of $w_q^\dagger(f_q(q))$ is read by process $f_q(q)$, the counter is gradually decremented to 0. Each intermediate process between q and p relays s every time it reads s. Thus the current content of q is relayed in the tree at least five times and p stores the content of p at least five times in $\omega_p^\ddagger$. (It may be relayed and stored more than five times if the



(a) (b)

Figure 5.8: Faults at intermediate processes on the *virtual link* (q, p)

content of ω_q remains unchanged.) □

Proposition 2 *If process p stores five s's in $\omega_p^{\ddagger}$, then p delivers s.*

Proof. Let c'_p be the content of c_p when the first s is stored in $\omega_p^{\ddagger}$. Majority is applied after $(c'_p + 1)^{th}$ s is stored in $\omega_p^{\ddagger}$. If $c'_p \leq 1$, then the next majority will be s since $(5 - (c'_p + 1))$ s's are stored in $\omega_p^{\ddagger}$. Else $(c'_p > 1)$, the current majority is s. □

When the majority is s, p delivers s. This corresponds to a situation in the ring where p reads its predecessor q's state s. Then p executes $Action(\omega_p, s)$ and updates ω_p with the resulting content. This corresponds to a situation in the ring where p executes local computation and writes to its local variables.

Lemma 15 *Starting from an arbitrary initial configuration, RET reaches in $O(\Delta)$ rounds (where Δ is the maximum degree of a node in the tree) a configuration such that Propositions 1 and 2 hold for every process in every configuration thereafter.*

Proof. Let p and q be two processes such that p is the successor of q. After process q has reset c_q to 4 at least once, ω_q changes when the process $f_q(q)$ has read the content of ω_q five times. Since p has reset c_p to 4, p applies majority once during the acquisition of s.

Propositions 1 and 2 hold for every process. In a finite time, each process executes C_1 or A_1 of Protocol 5.3.1. By using link alternator, a process communicates with one neighbor every $O(\Delta)$ rounds. Thus, $O(\Delta)$ rounds are necessary. □

Lemma 16 *If neither process q nor p is faulty, and $\omega_q = b$, then p delivers the value b if no fault occurs at the intermediate processes, otherwise the value b may be lost, but no fictitious data is delivered at p.*

Proof. The distance between q and p must be at least two, but no more than three. Let u be the next process after q on the tree towards p: $f_q(q) = u$. (Figure 5.8.)

Let a be the majority at p before the first b is stored at p. Let c be the next content of ω_q after b.

If none of the intermediate processes between q and p in the tree is faulty and $\omega_q = b$, then p stores five b's in $\omega_p^{\ddagger}$ (Proposition 1). Majority is applied at process p when c_p becomes 0. Data b is either the current, or the next majority (Proposition 2).

If u is faulty, and the fault at u affected $w_u^{\dagger}(f_u(q))$ such that $w_u^{\dagger}(f_u(q)) \neq w_q^{\dagger}(u)$, then data b that relayed to p from q through u is corrupted.

So, we have two cases:

Case A) The distance between q and p is two (Figure 5.8(a)).

By the communication synchronizer, after p stores a from u, u copies the correct data from q and corrects the entry in the content table. So, next time u relays to p correct data. So, instead of the sequence $b\ b\ b\ b\ b$, process p stores four b's and one faulty data f. We have then five cases, depending on the position of the faulty data.

1. Process p stores $f\ b\ b\ b\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(b)). If $c_p = 2$ when f is stored, then the next majority will be $\perp$, followed by c, and b is lost. Otherwise data b is delivered at p.
2. Process p stores $b\ f\ b\ b\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(c)). Same as Case 1.
3. Process p stores $b\ b\ f\ b\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(d)). Same as Case 1.
4. Process p stores $b\ b\ b\ f\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(e)). If $c_p = 1$ when the first b is stored, then the next majority will be $\perp$, followed by c. Thus b is lost. Otherwise b is delivered at p.
5. Process p stores $b\ b\ b\ b\ f$ in $\omega_p^{\ddagger}$ (Figure 5.9(f)). If $c_p = 1$ when the first b is stored then the next majority will be $\perp$, followed by c. Thus b is lost. Otherwise b is delivered at p.

Case B) The distance between q and p is three (Figure 5.8(b)).

Let u, v be the intermediate processes in the tree topology, and assume that either they are both faulty. After p stores a corrupted data from v, v copies another corrupted data from u that p also stores the next time p and v communicates. Then, u copies the correct data from q, so next time u relays to v the correct data.

So, instead of the sequence $b\ b\ b\ b\ b$, process p stores three b's and two consecutive faulty data f_1 and f_2. Let $c\ c\ c\ c\ c$ be the sequence to follow the sequence of b. We have five cases, depending on the position of the faulty data.

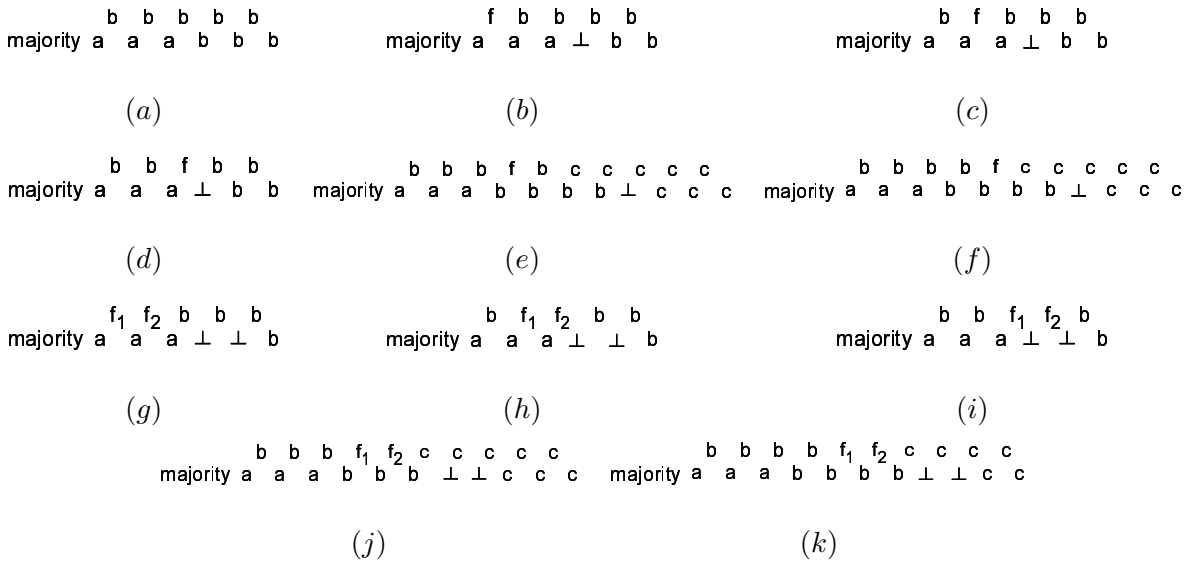


Figure 5.9: Majority values at process p

1. Process p stores $f_1\ f_2\ b\ b\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(g)). If $c_p \in \{2, 3\}$ when f_1 is stored, then the next majority will be $\perp$, followed by c. Thus b is lost. Otherwise b is delivered at p.
2. Process p stores $b\ f_1\ f_2\ b\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(h)). Same as Case 1.
3. Process p stores $b\ b\ f_1\ f_2\ b$ in $\omega_p^{\ddagger}$ (Figure 5.9(i)). Same as Case 1.
4. Process p stores $b\ b\ b\ f_1\ f_2$ in $\omega_p^{\ddagger}$ (Figure 5.9(j)). If $c_p \in \{0, 1\}$ when the first b is stored, then the next majority (after b's) will be $\perp$, followed by c. Thus b is lost. Otherwise b is delivered at p.
5. Process p stores $b\ b\ b\ b\ f_1\ f_2\ c\ c\ c\ c$ (Figure 5.9(k)). If $c_p \in \{1, 2\}$ when the first b is stored, then the next majority (after b's) will be $\perp$, followed by c. Thus b is lost. Otherwise b is delivered at p.

Consequently, if a fault occurred at some intermediate process(es) when p was about to collect five b's, in the worst case, instead of five b's, p will store three b's and two faulty data. Nevertheless, the corrupted data is not enough for a majority, thus data b can be either delivered at p or lost. □

Lemma 17 *If ω_q changes from a to b and the fault at p occurred after p stored five a's then data a and b may be lost, but no data relayed before a or after b is lost. The contents of p, ω_p, is corrupted at most once by the fault in the ring and no fictitious data is delivered at p instead of b.*

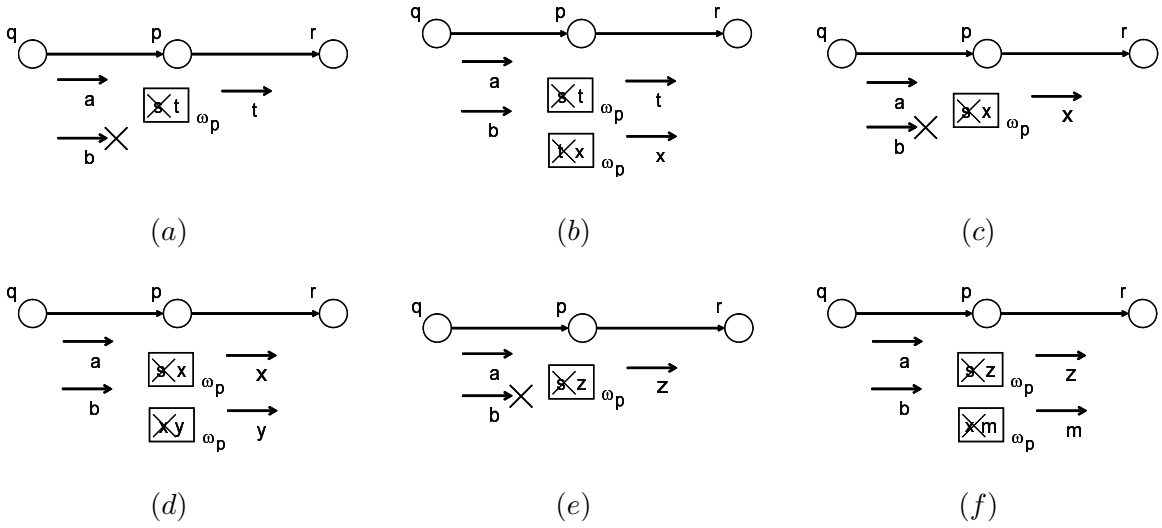


Figure 5.10: Fault at p in the virtual ring (Data a was delivered before the fault.)

Proof. Let n_b be the number of b's still left to be stored by p after the fault at p ($1 \leq n_b \leq 5$) and c'_p be the value of c_p before the fault.

After p stored five a's, p would receive at most two more b's before p delivers a. So, we have three cases depending on the value of c'_p.

1. $c'_p = 0$. If there was no fault, p would have computed majority after p stored one b and $\omega_p^{\ddagger} = \{a, a, a, a, b\}$ ($n_b = 5$) or $\{a, a, a, b, b\}$ ($n_b = 4$). Thus, a was not delivered at p before the fault.

2. $c'_p = 1$. If there was no fault, p would have computed majority after p stored two b's and $\omega_p^{\ddagger} = \{a, a, a, b, b\}$ ($n_b = 5$). Thus, a was not delivered at p before the fault.

3. $c'_p > 1$ The majority has already computed on $\omega_p^{\ddagger}$ that contains at least three a's and a was delivered at p.

We consider two cases: Case A) when a was delivered at p before the fault and Case B) when a was not delivered at p before the fault.

Case A) Data a is delivered at p before the fault

Let the fault changes ω_p from s to t. We have five cases, depending on the value of c_p after the fault.

Case $c_p = 0$**.** A majority function is applied to $w_p^{\ddagger}$ after one b is stored in $\omega_p^{\ddagger}$.

Let $w = maj(\{b, w_1, \ldots, w_4\}, 5)$. We have three cases, depending on the value of w:

1. $w = \perp$. Thus ω_p remains unchanged (value t), and r stores five t's from p. By Proposition 2, t is delivered at r.
   If $n_b < 4$ then p will store at most two other b's (since one b was already stored), thus too few b's to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to t, and caused b to be lost (Figure 5.10(a)).
   Else ($n_b \geq 4$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to t, but p reads b (Figure 5.10(b)).
2. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r.
   If $n_b < 4$ then p will store too few b to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p changed ω_p from s to x, and caused b to be lost (Figure 5.10(c)).
   Else ($n_b \geq 4$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $y = Action(x, b)$, and r stores five m's. By Proposition 2, y is delivered at r. This corresponds in the virtual ring as a fault at process p changed ω_p from s to x, but p reads b (Figure 5.10(d)).
3. $w \neq \perp \wedge w \neq b$. Then ω_p is changed to $z = Action(t, w)$, and r stores five z's. By Proposition 2, z is delivered at r.
   If $n_b < 4$ then p will store too few b to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p changed ω_p from s to z, and caused b to be lost (Figure 5.10(e)).
   Else ($n_b \geq 4$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $m = Action(z, b)$, and r stores five m's. By Proposition 2, m is delivered at r. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to z, but p reads b (Figure 5.10(f)).

Case $c_p = 1$**.** Thus r reads one t from p.

If $n_b = 1$ then p stores one b and after that p stores another data q relays after b. Let c be that data and $w = maj(\{c, b, w_1, w_2, w_3\}, 5)$. Based on w, we have three cases.

1. $w = \perp$. Thus ω_p remains unchanged (value t), and r stores five t's. By Proposition 2, t is delivered at r. The next majority will be c and ω_p is changed to $Action(t, c)$. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to t and caused b to be lost (Figure 5.10(a)).

2. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r. The next majority will be c. This corresponds in the virtual ring as a fault at process p changed ω_p from s to x and caused b to be lost (Figure 5.10(c)).
3. $w \neq \perp \wedge w \neq b$. Then ω_p is changed to $z = Action(t, w)$, and r stores five z's. By Proposition 2, z is delivered at r. The next majority will be c. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to z and caused b to be lost (Figure 5.10(e)).

If $n_b > 1$, let $w = maj(\{b, b, w_1, w_2, w_3\}, 5)$. Based on w, we have three cases.

1. $w = \perp$. Thus ω_p remains unchanged (value t), and r stores five t's. By Proposition 2, t is delivered at r.
   If $1 < n_b < 5$ then p will store at most two more b's from q (already two b's have been stored) thus too few b's to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to t, and caused b to be lost (Figure 5.10(a)).
   Else ($n_b = 5$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r. Since one t is not enough to be majority at r, this corresponds in the virtual ring as a fault at process p changed ω_p from s to x, and caused b to be lost (Figure 5.10(c)).
2. $w \neq \perp \wedge w \neq b$. Then ω_p is changed to $z = Action(t, w)$, and r stores five z's. By Proposition 2, z is delivered at r.
   If $1 < n_b < 5$ then p will store too few b to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to z, and caused b to be lost (Figure 5.10(e)).
   Else ($n_b = 5$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $m = Action(z, b)$, and r stores five m's. By Proposition 2, m is delivered at r. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to z, but p reads b (Figure 5.10(f)).
3. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r.
   If $1 < n_b < 5$ then p will store too few b to be able to be the next majority at p. Thus b is lost. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to x, and caused b to be lost (Figure 5.10(c)).
   Else ($n_b = 5$) then the next majority at p is b. Thus b is delivered at p, ω_p is

changed to $y = Action(x, b)$, and r stores five y's. By Proposition 2, y is delivered at r. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to x, but p reads b (Figure 5.10(d)).

Case $c_p = 2$**.** Thus r stores two t's from p.

If $n_b < 3$ then p stores at least one b and after that p stores another data q relays after b. Let c be that data and $w = maj(\{c, c, b, w_1, w_2\}, 5)$. Based on w we have three cases.

1. $w = \perp$. Thus ω_p remains unchanged (value t), and r stores five t's. By Proposition 2, t is delivered at r. The next majority will be c and ω_p is changed to $Action(t, c)$. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to t and caused b to be lost (Figure 5.10(a)).
2. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r. The next majority will be c. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to x and caused b to be lost (Figure 5.10(c)).
3. $w \neq \perp \wedge w \neq b$. Then ω_p is changed to $z = Action(t, w)$, and r stores five z's. By Proposition 2, z is delivered at r. The next majority will be c. This corresponds in the virtual ring as a fault at process p that changed ω_p from s to z and caused b to be lost (Figure 5.10(e)).

Else ($n_b \geq 3$) then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $x = Action(t, b)$, and r stores five x's. By Proposition 2, x is delivered at r. Since two t's is not enough to make majority at r, this corresponds in the virtual ring as a fault at process p that changed ω_p from s to x, and caused b to be lost (Figure 5.10(c)).

Case $c_p = 3$**.** Thus r stores three t's from p. Then follow the same as Case $c_p = 2$.

Case $c_p = 4$**.** Thus r stores four t's from r. Then follow the same as Case $c_p = 2$.

Case B) Data a is not delivered at p before the fault.

Thus, $n_b \geq 4$. Let the fault changes ω_p from s' to t'. We have five cases, depending on the value of c_p after the fault.

Case $c_p = 0$**.** A majority function is applied to $w_p^{\ddagger}$ after one b is stored in $\omega_p^{\ddagger}$.

Let $w = maj(\{b, w_1, \ldots, w_4\}, 5)$. We have three cases, depending on the value of w:

1. $w = \perp$. Thus ω_p remains unchanged (value t'), and r stores five t' from p. By Proposition 2, data t' is delivered at r. Since $n_b \geq 4$, the next majority at p is

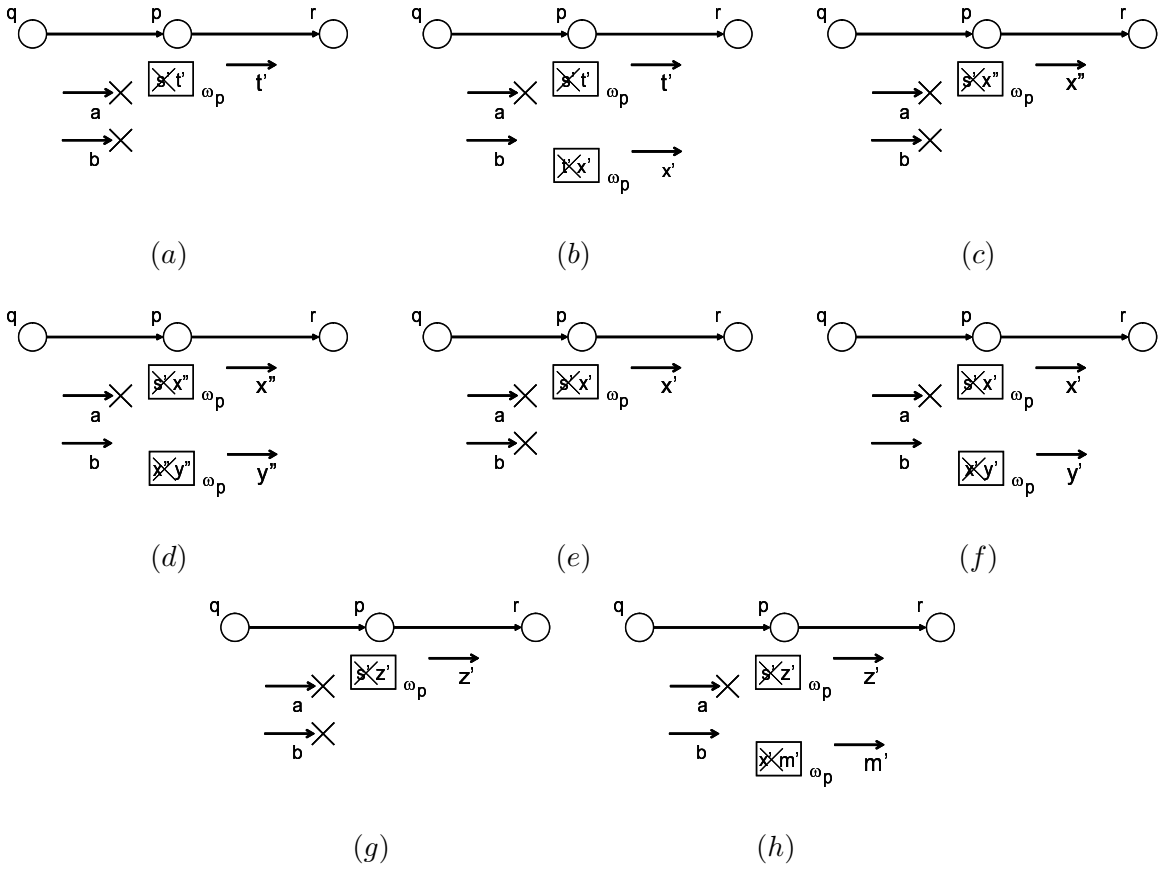


Figure 5.11: Fault at p in the virtual ring (Data a was not delivered before the fault.)

b. Thus b is delivered at p, ω_p is changed to $x' = Action(t', b)$, and r stores five x'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to t', and caused a to be lost but p reads b. (Figure 5.11(b)).

2. $w \neq \perp \wedge w = a$. Then ω_p is changed to $x'' = Action(t', a)$, and r stores five x''. By Proposition 2, data x'' is delivered at r. Since $n_b \geq 4$, the next majority at p is b. Thus b is delivered at p, ω_p is changed to $y'' = Action(x'', b)$, and r stores five y''. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x'', and caused a to be lost but p reads b (Figure 5.11(d)).
3. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x' = Action(t', b)$, and r stores five x'. By Proposition 2, data x' is delivered at r. Since $n_b \geq 4$, the next majority at p is b. Thus b is delivered at p, ω_p is changed to $y' = Action(x', b)$, and r stores five y'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x', and caused a to be lost but p reads b (Figure 5.11(f)).
4. $w \neq \perp \wedge w \neq a \wedge w \neq b$. Then ω_p is changed to $z' = Action(t', w)$, and r stores five z'. By Proposition 2, data z' is delivered at r. Since $n_b \geq 4$, the next majority at p is b. Thus b is delivered at p, ω_p is changed to $m' = Action(z', b)$, and r stores five m'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to z', and caused a to be lost but p reads b (Figure 5.11(h)).

Case $c_p = 1$**.** Thus r reads one t from p.

Since $n_b \geq 4$, let $w = maj(\{b, b, w_1, w_2, w_3\}, 5)$. Depending on w, we have three cases.

1. $w = \perp$. Thus ω_p remains unchanged (value t'), and r stores five t'. By Proposition 2, data t' is delivered at r. Next two cases can occur:
   (i) If $4 \leq n_b < 5$ then follow the same as Case A. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to t', and caused data a and b to be lost (Figure 5.11(a)).
   (ii) If $n_b = 5$ then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $x' = Action(t', b)$, and r stores five x'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x', and caused a to be lost but p reads a (Figure 5.11(b)).
2. $w \neq \perp \wedge w = a$. Then ω_p is changed to $x'' = Action(t', a)$, and r stores five x''. By Proposition 2, data x'' is delivered at r. Next two cases can occur:
   (i) If $4 \leq n_b < 5$ then follow the same as Case A. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x'', and caused a and b to be lost (Figure 5.11(c)).
   (ii) If $n_b = 5$ then the next majority at p is b. Thus b is delivered at p, ω_p is

changed to $y'' = Action(x'', b)$, and r stores five y''. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x'', and caused a to be lost but p reads b (Figure 5.11(d)).

3. $w \neq \perp \wedge w = b$. Then ω_p is changed to $x' = Action(t', b)$, and r stores five x'. By Proposition 2, data x' is delivered at r. Next two cases can occur:
   (i) If $4 \leq n_b < 5$ then follow the same as Case A. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x', and caused a and b to be lost (Figure 5.11(e)).
   (ii) If $n_b = 5$ then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $y' = Action(x', b)$, and r stores five y'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x', and caused a to be lost but p reads b (Figure 5.11(f)).
4. $w \neq \perp \wedge w \neq a \wedge w \neq b$. Then ω_p is changed to $z' = Action(t', w)$, and r stores five z'. By Proposition 2, data z' is delivered at r. Next two cases can occur:
   (i) If $4 \leq n_b < 5$ then follow the same as Case A. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to z', and caused a and b to be lost (Figure 5.11(g)).
   (ii) If $n_b = 5$ then the next majority at p is b. Thus b is delivered at p, ω_p is changed to $m' = Action(z', b)$, and r stores five m'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to z', and caused a to be lost but p reads b (Figure 5.11(h)).

Case $c_p = 2$**.** Thus r stores two t's from p. Since $n_b \geq 4$, the next majority at p is b. Thus b is delivered at p, ω_p is changed to $x' = Action(t', b)$, and r stores five x'. This corresponds in the virtual ring as a fault at process p changed ω_p from s' to x', and caused a and b to be lost (Figure 5.11(e)).

Case $c_p = 3$**.** Thus r stores three t's from p. Then follow the same as Case $c_p = 2$.

Case $c_p = 4$**.** Thus r stores four t's from r. Then follow the same as Case $c_p = 2$.

□

Lemma 18 *If q changes ω_q from b to c after q changes ω_q from a to b and the fault at p and intermediate processes occurred after p stored five a's then data a, b, and c may be lost, but no data relayed before a or after c is lost. The contents of p, ω_p, is corrupted at most once by the fault in the ring and no fictitious data is delivered at p.*

Proof. From Lemma 16 the data relayed when the fault corrupts the intermediate processes can be lost. During the acquisition of b's at p, data b or c is relayed and can be corrupted. From

Lemma 17 data a and b can be lost by the fault at p. Thus, at most three messages are lost by the fault at p and intermediate processes. □

Theorem 5 *Protocol RET provides a causal simulation of a ring protocol executed in a tree, and also preserves the self-stabilization and the fault-containment of the original ring protocol.*

Proof. Let $\mathcal{A}$ be a protocol on a ring and ω_p be the local state of some process p in $\mathcal{A}$. The set of local variables at p in *RET* consists of ω_p, c_p, $w_p^\dagger$, and $w_p^\ddagger$. The condition of Definition 10 is satisfied: From Lemma 15 the system eventually reaches a configuration σ such that after σ Proposition 1 and 2 holds. In $\mathcal{A}$, if a process p reads the state a of its predecessor q, executes local computation and writes its local variables (updates ω_p), then in *RET* the following sequence of execution steps occurs: a is delivered at p and p executes $Action(\omega_p, a)$ and writes the results to ω_p. From Proposition 1 and 2, this always holds after σ if no fault occurs.

From Lemma 16, 17, and 18, when some fault corrupts p and intermediate processes on the virtual link towards p, at most three data relayed to p is lost and at most one data for each neighbor of p is lost. A lost data corresponds to a state of some process in the virtual ring that was not read by its successor. Thus, we conclude that the condition of Definition 11 is satisfied. □

5.3.3 Performance Evaluation

The execution of ring protocols in the virtual ring is slowed down because the dilation of *vertex bijective ring* is three and the communication synchronization mechanism for *RET* forces a process to communicate with its neighbors in a specific order. The *slowdown* of protocol P_r for a simulated protocol P_v is the maximum number of rounds in P_r that are necessary for one read action of P_v. Since the dilation is constant, we show the slowdown of *RET* is proportional to the maximum degree of some process in the tree.

Theorem 6 *Slowdown of RET for a ring protocol executed in a tree of maximum degree* Δ *is* 8Δ.

Proof. Let p and q be neighboring processes in the tree such that $f_p(q) = p$. After p communicates with q, the synchronization mechanism forces p to communicate with other neighbors than q before communicating with q again. In [45] it is proved that starting from any arbitrary initial configuration, after Δd rounds where d is the diameter of the graph, it is guaranteed that for any process p, p can communicate with its neighbor q k times in $k\Delta$ rounds.

One virtual link consists of at most three links of the tree. Thus, at most $3k\Delta$ rounds are necessary to relay k pieces of data between endpoint processes. For one piece of data to be

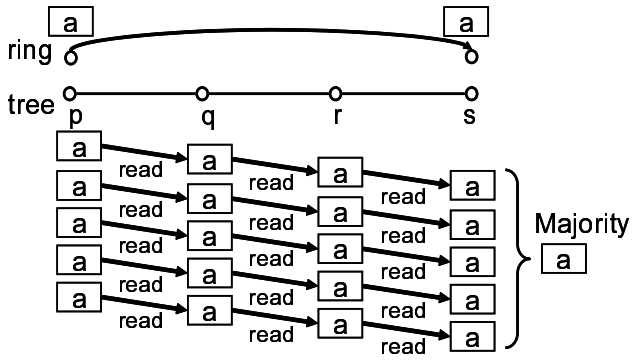


Figure 5.12: The delay caused by *RET*

delivered at an endpoint process, five pieces of identical data should be relayed and the data relayed is pipelined (Figure 5.12). Thus, for one data to be relayed, $(5+3)\Delta$ rounds is necessary and the slowdown of *RET* is 8Δ. □

5.4 Example of 1-fault-containing Leader Election

We show how protocol RET can be used to design a 1-fault-containing leader election protocol in arbitrary trees from the 1-fault-containing leader election protocol of Ghosh and Gupta [22] on bidirectional rings.

Let *LE* be the Ghosh and Gupta's leader election protocol that selects the node with the maximum ID as the leader. We present a causal simulation of *LE* on arbitrary trees that is 1-fault-containing and is obtained by combining *RET* and *LE*.

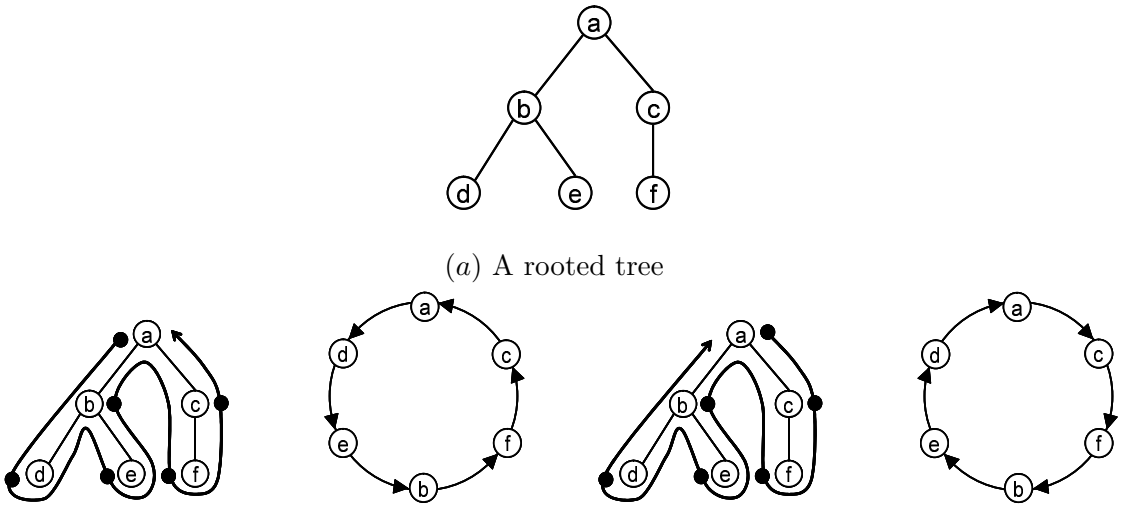
We denote the predecessor of p with pre_p.

In *LE* each process p has a unique ID id_p, and other variables as follows:

1. max_p is the maximum ID known by p.
2. $dist_p$ is the distance from the leader on the counter-clockwise ring.
3. $q_p \in \{0, 1\}$ indicates whether p asked the predecessor pre_p ($q_p = 1$) if pre_p has to change max_{pre_p} or dis_{dist_p}.
4. $a_p \in \{0, 1, \perp\}$ indicates whether p answered the question from the successor; $a_p = 1$ (respectively, 0) if it has (respectively, does not have) to change max_p or $dist_p$; if it has not answered then $a_p = \perp$.
5. c_p stores a copy of a_{pre_p}

Let q be the process with the maximum ID among all processes in V, and let K be the value of id_q.

In a legitimate configuration of *LE*, the following conditions hold:



(a) A rooted tree

(b) Forward ring

(c) Backward ring

Figure 5.13: Embedding forward and backward rings

(i) For all process $p \in V$, $max_p = K$, and

(ii) $dist_q = 0$ and for any other process p, $dist_p = dist_{pre_p} + 1$.

LE is designed for bidirectional rings and RET for unidirectional rings. RET uses the routing function from Section 5.3 on what we call the *forward* ring (counter-clockwise preorder-postorder traversal). The reverse of the routing function provides communication on what we call the *backward* ring (clockwise preorder-postorder traversal).

Each process p has to evaluate the guards of LE and execute the corresponding actions with the value that p has just delivered in the virtual ring. The guards of LE contains the state of p's predecessor and successor. However, when process p executes $Action()$ with RET^+, p has delivered the data just from its predecessor.

To evaluate and execute the guarded actions of LE with the latest data that p has just delivered from its predecessor and successor, we divide LE into two distinct set of guarded commands, LE^+ and LE^-. The guards and actions in LE^+ at process p contains just p's local variables and its predecessor's variables while the guards and actions in LE^- contains just p's local variables and its successor's variables. We obtain LE by the union of LE^+ and LE^-. The guarded commands in LE^+ are executed when RET^+ executes $Action()$ and the guarded commands in LE^- are executed when RET^- executes $Action()$.

While RET^+ and RET^- use different counters, cache tables, and contents tables, the contents of both rings are common: at process p, $\omega_p^+ = \omega_p^- = (id_p, max_p, dist_p, q_p, a_p, c_p)$. Whenever the distributed daemon selects process p, RET^+ and RET^- are executed consecutively. When RET^+ (respectively, RET^-) executes $Action()$, LE^+ (respectively, LE^-) is executed. We call

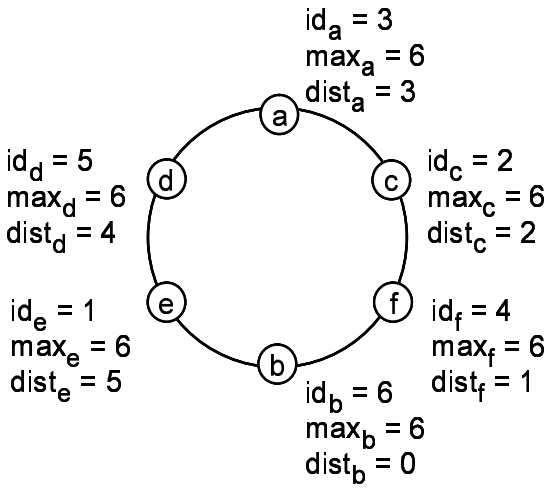


Figure 5.14: Leader election in the bidirectional ring

this implementation as *BRET* (Bidirectional *RET*).

In *BRET*, a selected process p may execute at most two actions of *LE*: one is executed by RET^+ and the other is executed by RET^-. This corresponds to the case that p is selected by distributed daemon successively in the original execution of *LE* on a bidirectional ring.

In other cases, p may execute the actions of LE^+ several times without executing LE^-. Since we adopt the model that allows an enabled process to execute one of the enabled actions, this corresponds to the case that p does not select the enabled guard of LE^-.

The following lemma holds immediately.

Lemma 19 *For any execution E_r of BRET, there is an execution E_v of LE such that E_r is obtained by a causal shift of E_v.*

Thus, *BRET* provides a causal simulation of *LE*.

For example, consider the case where *BRET* is executed on the tree in Figure 5.13(a). The forward and the backward ring of the tree are shown in Figure 5.13(b) and 5.13(c). The forward ring enables each process to read its predecessor's state and the backward ring enables each process to read its successor's state. The predecessor for process a is c (Figure 5.13(b)) and the successor for a is d (Figure 5.13(e)).

Figure 5.14 shows a legitimate configuration of the leader election when $id_a = 3$, $id_b = 6$, $id_c = 2$, $id_d = 5$, $id_e = 1$, and $id_f = 4$. Let process c be corrupted by a fault and after the fault, the contents of c, ω_c is $id_c = 2$, $max_c = 5$, $dist_c = 0$, $q_c = 0$, $a_c = \perp$, $c_c = 0$ and we denote this by $(2, 5, 0, 0, \perp, 0)$.

Clearly, not only the state of process c in the virtual ring but also the data in the contents

table at c can be corrupted. Because c is on the virtual link (b, f) of the forward ring and on the virtual link (f, b) of the backward ring, process b and f can read at most one corrupted data. This is eliminated by the majority computation at each process.

The corruption at c makes c to set $q_c = 1$ and a to set $q_a = 1$. No other processes change their question flags. This is because each process checks the state of the predecessor first. In this case, $max_b \neq max_c$ at process c and $max_c \neq max_a$ at process a.

Let us concentrate on the forward ring. If the fault changes $c_c^+ = 3$, then c executes *BRET* after it reads the content at f once. When the majority of the cache table may be different from the content at f, c may change its content with incorrect data: e.g. $(2, 5, 0, 0, \perp, 1)$. The successor of c, a reads the content $(2, 5, 0, 0, \perp, 0)$ just once and the state cannot be delivered at a. Thus, $(2, 5, 0, 0, \perp, 0)$ can be ignored in the causal simulation and for a, the fault seems to change the contents at c to $(2, 5, 0, 0, \perp, 1)$. After that, c corrects correct data from f and executes *BRET* and executes recovery actions.

5.5 Concluding Remarks

In this chapter, we proposed protocol *RET* that preserves the fault-containment property of a ring protocol executed on an arbitrary rooted tree. Our protocol ensures that along any link of a virtual ring embedded on a tree, there is no data corruption, neither data creation. Because the delay of each virtual link differs from others, *RET* cannot trace the global configurations of the original execution. We introduced causal simulation that preserves the read/write causality of the original execution. Causal simulation is strong enough to execute the same task as the original protocol for any reactive and non-reactive tasks as long as the safety property of the task is defined by the read/write causality. Because the safety properties of many reactive and non-reactive tasks are defined by the read/write causality, the proposed protocol *RET* is useful in extending the application of existing fault-containing ring protocols.

Though protocol *RET* is designed in the locally shared memory model, it can be extended to ring protocols written in message-passing model by considering a time-stamp to the data sent along the virtual link embedded on a tree. The time-stamp will takes integer values in the range $1 \ldots 5$. Also, the proposed communication mechanism can be used to embed virtual links of other topology embeddings.

Chapter 6

Conclusion

6.1 Summary of the Results

In this dissertation, we focused on hierarchical design of fault-containing self-stabilizing protocols. Hierarchical structure of protocols facilitates the design of new protocols and extends the application of existing protocols. Fault-containing protocols have the power of adaptive self-stabilization, i.e. they provide self-stabilization for large scale faults and fault-containment for small scale faults. This adaptability is useful in practice because in real networks, catastrophic faults rarely occur while small scale faults are more likely to occur frequently.

In Chapter 3 and Chapter 4, we proposed hierarchical composition techniques for fault-containing protocols. The goal of the composition techniques is to preserve the fault-containment property of source protocols. Our strategy *RWFC* is to control the execution of source protocols and we utilized the fault-containment property of source protocols to control their execution. In Chapter 3, we utilized the temporal containment property of source protocols and, as a component of the composition protocol *RWFC-LNS*, we designed local neighborhood synchronizer *LNS* that synchronizes only small number of processes around faulty processes for a short period of time after a fault. In Chapter 4, we utilized the spatial containment property of source protocols and, as a component of the composition protocol *RWFC-IcD*, we designed inconsistency detector *IcD* that checks the inconsistency of source protocols. Though the component protocols (*LNS* and *IcD*) impose some overhead on the composite protocol, the overhead is bounded by the containment property of source protocols. Hence, the composite protocol preserves the spatial and/or temporal containment property of source protocols at the cost of small overhead. The composition framework for fault-containing protocols is important both theoretically and practically. We can design new fault-containing protocols easily at the top of existing fault-containing protocols with the proposed composition framework. This is the first step in facilitating the

design of new fault-tolerant distributed protocols.

In Chapter 5, we introduced ring embedding on an arbitrary rooted tree and proposed causal simulation technique for fault-containing ring protocols on the embedded ring. The proposed protocol *RET* embeds virtual links on the real topology and in the embedded ring there is neither data corruption nor data creation. Because the delay of each virtual link is different from others, *RET* preserves read/write causality of original executions that is strong enough to guarantee that the simulation executes the same task as the original execution. We call the execution of the source protocol on the embedded ring causal simulation. The slowdown of the simulation depends on the dilation of the embedding and our ring embedding has the dilation of three. Hence, the simulation protocol simulates fault-containing ring protocols on a rooted tree with a small slowdown. The proposed topology embedding demonstrates the possibility of uniform framework based on topology embedding that extends the application of existing fault-containing protocols.

6.2 Future Directions

Regarding the proposed methods for hierarchical design of fault-containing protocols, there exist many issues of both theoretical interest and practical interest.

In Chapter 3 and Chapter 4, we discussed hierarchical composition of self-stabilizing protocols that preserves the fault-containment property. The point is how to guarantee the recovery of the lower protocol when the upper protocol starts its execution. In the proposed composition technique, we utilize the containment property of fault-containing protocols to control the execution of source protocols, i.e. temporal containment property and spatial containment property. Fault-containing protocols form one subclass of adaptive self-stabilizing protocols. One extension of our work is to propose hierarchical composition technique for other adaptive self-stabilizing protocols preserving their own adaptability, e.g. time-adaptive stabilization, superstabilization, local stabilization, and time-to-fault adaptive stabilization. To develop hierarchical composition for adaptive self-stabilizing protocols, the adaptability of source protocols can be used to guarantee the recovery of the lower protocol.

Another interesting issue in composing adaptive self-stabilizing protocols is to investigate the trade-off between the adaptability preserved in the composite protocol and the cost (e.g. time and space) paid to preserve the adaptability of source protocols. The proposed composition techniques guarantee that the lower protocol has recovered when the upper protocol starts its execution. However, we can relax the condition by allowing the upper protocol to be executed after some safety property holds in the lower protocol. This relaxed condition can make the

composite protocol satisfy some safety property quickly. However, the critical issue is how to bound the spread of the effect of faults.

In Chapter 5, we introduced one-to-one ring embedding on an arbitrary rooted tree and proposed a simulation technique that preserves the fault-containment property simulated on the embedded ring. We proposed a communication mechanism that realizes virtual links on the real topology. Though this communication mechanism is implemented for ring embeddings, we can apply this mechanism to any virtual link embedding with constant dilation. For example, in a network clustering based on a maximal independent set, the maximum distance between cluster heads is three and we can easily apply the proposed communication mechanism to realize almost reliable communication between cluster heads. It is one of the interesting extensions for our communication mechanism to seek for applications in real networks.

Another future work is to relax the communication model and fault model. Though the proposed simulation technique is designed for locally shared memory model, the technique also fits the message passing model. This is because each virtual link has different but bounded delay that corresponds to message passing model and channels with bounded delay. Fault tolerance against Byzantine faults is well studied in the area of self-stabilization [5, 44, 46, 54]. By embedding multiple disjoint virtual links between two adjacent virtual processes, we can realize communication mechanism with Byzantine fault tolerance. However, the problem is the communication slowdown because the slowdown of the proposed communication mechanism depends on the number of virtual links. Thus, the message duplication technique itself needs to be improved for multiple virtual links. Adding Byzantine fault tolerance should be significant improvement for our simulation technique because Byzantine fault is the strongest fault model.

To develop further methods for hierarchical structures of fault-tolerant distributed protocols, it is important to examine other useful properties for implementing hierarchical structures. For example, consider the stability in output of non-masking fault-tolerant protocols that guarantees even when some faults or topology changes occur, the output of a protocol may change to adopt them but the changes in the output are as small as possible. This stability in output can prevent the application of the protocol from changing its configuration frequently according to the unnecessary changes of the input. Adding these useful properties to existing fault-tolerant protocols is important both theoretically and practically.

Acknowledgments

The author has been fortunate to receive assistance from many people. She would especially like to express her gratitude to her supervisor Professor Toshimitsu Masuzawa for his guidance and encouragement. The author has also received precious advice from Professors of the Graduate School of Information Science and Technology, Osaka University. Among them, the author would like to extend her gratitude to Professor Kenichi Hagihara, Professor Katsuro Inoue, and Professor Hirotsugu Kakugawa for their valuable comments on this dissertation. The author would like to acknowledge Professor Yasushi Yagi and Professor Shinji Kusumoto for their helpful comments on her work.

The author would like to thank Professor Koichi Wada at Nagoya Institute of Technology, Professor Akinori Saitoh at Tottori University of Environmental Study, Professor Hideo Masuda at Kyoto Institute of Technology, Dr. Tadashi Araragi at NTT Communication Science Laboratories, Associate Professor Yoshiaki Katayama at Nagoya Institute of Technology, Professor Sébastien Tixeuil at Université Pierre et Marie Curie, Research Associate Doina Bein at Pennsylvania State University, Assistant Professor Sayaka Kamei at Hiroshima University, and Assistant Professor Taisuke Izumi at Nagoya Institute of Technology for their useful comments on her work. The author also thanks to Mrs. Tomoko Arakawa, Mrs. Megumi Kunimatsu, and Ms. Fusami Nagae for their kind support. She is also grateful to the staffs and students of Algorithm Engineering Laboratory, the Graduate School of Information Science and Technology, Osaka University. In particular, she thanks to Assistant Professor Fukuhito Ooshita, Dr. Yoshihiro Nakaminami, and Dr. Tomoko Izumi for their time and kindness.

Finally, the author wishes to thank her parents Masayuki Yamauchi and Chieko Yamauchi, and all of her families for their support and kindness during her life at the university.

Bibliography

[1] Y. Afek and S. Dolev. Local stabilizer. In *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems*, pages 74–84, June 1997.

[2] A. Arora and A. Singhai. Fault-tolerant reconfiguration of trees and rings in networks. In *Proceedings of the 2nd International Conference on Network Protocols*, pages 221–228, Oct. 1994.

[3] A. Arora and H. Zhang. LSRP: Local stabilization in shortest path routing. In *Proceedings of the International Conference of Dependable Systems and Networks*, pages 139–148, June 2003.

[4] Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 33–42, July 2003.

[5] F. Bastani, I. Yen, and Y. Zao. On self-stabilization, non-determinism and inherent fault tolerance. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, 1989.

[6] J. Beauquier, M. Gradinariu, and C. Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems*, pages 19–34, Oct. 2001.

[7] D. Bein, A. K. Datta, and L. L. Larmore. Self-stabilizing space optimal synchronization algorithm on trees. In *Proceedings of the 13th Colloquium on Structural Information and Communication Complexity*, pages 334–348, July 2006.

[8] D. Bein, A. K. Datta, and V. Villain. Self-stabilizing local routing in ad hoc networks. *The Computer Journal*, 50(2):197–203, 2007.

[9] S. C. Bruell, S. Ghosh, M. Karaata, and S. V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM Journal of Computing*, 29:600–614, 1999.

[10] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the 4th Workshop on Self-Stabilizing Systems*, pages 78–85, June 1999.

[11] N. Chen, H. Yu, and S. Huang. A self-stabilizing algorithm for constructing a spanning tree. *Information Processing Letters*, 39(3):147–151, 1991.

[12] A. Cournier, A. K. Datta, F. Petit, and V. Villain. Snap-stabilizing PIF algorithm in arbitrary networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 199–208, July 2002.

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

[14] S. Dolev. *Self-Stabilization.* MIT Press, Cambridge, MA, 2000.

[15] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, 1995.

[16] S. Dolev and T. Herman. Parallel composition for time-to-fault adaptive stabilization. *Distributed Computing*, 20:29–38, 2007.

[17] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, 1989.

[18] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message driven protocols. In *Proceedings of the 10th Annual ACM symposium on principles of distributed computing*, pages 281–293, 1991.

[19] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[20] F. C. Gartner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technology (PEFL), School of Computer and Communication Science, June 2003.

[21] S. Ghosh. *Distributed systems: an algorithmic approach.* Chapman & Hall/CRC, 2007.

[22] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[23] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, May 1996.

[24] S. Ghosh, A. Gupta, and S. Pemmaraju. Fault-containing network protocols. In *Proceedings of the 12th ACM Symposium on Applied Computing*, pages 431–437, Feb. 1997.

[25] S. Ghosh and X. He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73:145–151, 2000.

[26] S. Ghosh and S. V. Pemmaraju. Trade-offs in fault-containing self-stabilization. In *Proceedings of the 3rd Workshop on Self-stabilizing Systems*, pages 157–169, Aug. 1997.

[27] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on software engineering*, 17(9):911–921, 1991.

[28] T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *Proceedings of the 1st International Workshop of Algorithmic Aspects of Wireless Sensor Networks*, pages 45–58, July 2004.

[29] S. H. Hsu and S. T. Huang. A self-stabilizing algorithm for maximal matching. *Information processing letters*, 43:77–81, 1992.

[30] S. T. Huang and N. S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

[31] S. T. Huang and N. S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[32] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree network. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 487–494, May 1999.

[33] Y. Katayama and T. Masuzawa. A fault-containing self-stabilizing protocol for constructing a minimum spanning tree. *Transactions of the IEICE (In Japanese)*, J-84-D-I(9):1307–1317, 2001.

[34] Y. Katayama, E. Ueda, H. Fujiwara, and T. Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. *Journal of Parallel and Distributed Computing*, 62(5):865–884, 2002.

[35] S. S. Kulkarni and M. U. Arumugam. Transformations for write-all-with-collision model. In *Proceedings of the 7th International Conference of Principles of Distributed Systems*, pages 184–197, Dec. 2003.

[36] S. Kutten and B. Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 149–158, 1997.

[37] F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA 94403, USA, 1992.

[38] G. LeLann. Distributed systems: Towards a formal approach. In *Proceedings of the IFIP Congress '77*, pages 155–160, Aug. 1977.

[39] J. Lin and T. C. Huang. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. *IEEE Transactions on Parallel and Distributed Systems*, 14:742–754, 2003.

[40] X. Lin and S. Ghosh. Maxima finding in a ring. In *Proceedings of the 28th Annual Allerton Conference on Computers, Communication and Control*, pages 662–671, 1991.

[41] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[42] F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 383–393, Nov. 2007.

[43] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. In *Proceedings of the 14th Colloquium on Structural Information and Communication Complexity*, pages 96–108, June 2007.

[44] T. Masuzawa and S. Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. In *Proceedings of the 9th International Conference of Principles of Distributed Systems*, pages 118–219, Dec. 2005.

[45] Y. Nakaminami, T. Masuzawa, and T. Herman. Self-stabilizing agent traversal on tree networks. *IEICE Transactions of Information and Systems*, E87-D(12):2773–2780, 2004.

[46] M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, pages 22–29, Oct. 2002.

[47] T. Nolte and N. Lynch. A virtual node-based tracking algorithm for mobile networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, June 2007.

[48] M. Schneider. Self-stabilization. *ACM Computing Survey*, 25(1):45–67, Mar. 1993.

[49] M. Sekanina. On the ordering of the set of vertices of a connected graph. *Publications of the Faculty of Science, University of Brno*, 412:137–142, 1960.

[50] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Observation on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the 2nd Workshop on Self-stabilizing Systems*, May 1995.

[51] G. Tel. *Introduction to distributed algorithms.* Cambridge Univ. Press, Cambridge, U.K., 2nd edition, 2000.

[52] F. Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.

[53] K. Yoshida, H. Kakugawa, and T. Masuzawa. Observation on light weight implementation of self-stabilizing node clustering algorithms in sensor networks. In *Proceedings of the International Association of Science and Technology for Development International Conference on Sensor Networks*, pages 1–8, Sept. 2008.

[54] Y. Zhao and F. B. Bastani. A self-adjusting algorithm for byzantine agreement. *Distributed Computing*, 5:219–226, 1992.