USING EVENTUAL CONSISTENCY TO IMPROVE THE PERFORMANCE OF DISTRIBUTED GRAPH COMPUTATION IN KEY-VALUE STORES

By

Duong Ngoc Nguyen

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2021

ABSTRACT

USING EVENTUAL CONSISTENCY TO IMPROVE THE PERFORMANCE OF DISTRIBUTED GRAPH COMPUTATION IN KEY-VALUE STORES

By

Duong Ngoc Nguyen

Key-value stores have gained increasing popularity due to their fast performance and simple data model. A key-value store usually consists of multiple replicas located in different geographical regions to provide higher availability and fault tolerance. Consequently, a protocol is employed to ensure that data are consistent across the replicas. The CAP theorem states the impossibility of simultaneously achieving three desirable properties in a distributed system, namely consistency, availability, and network partition tolerance. Since failures are a norm in distributed systems and the capability to maintain the service at an acceptable level in the presence of failures is a critical dependability and business requirement of any system, the partition tolerance property is a necessity. Consequently, the trade-off between consistency and availability (performance) is inevitable. Strong consistency is attained at the cost of slow performance and fast performance is attained at the cost of weak consistency, resulting in a spectrum of consistency models suitable for different needs. Among the consistency models, sequential consistency and eventual consistency are two common ones. The former is easier to program with but suffers from poor performance whereas the latter suffers from potential data anomalies while providing higher performance.

In this dissertation, we focus on the problem of what a designer should do if he/she is asked to solve a problem on a key-value store that provides eventual consistency. Specifically, we are interested in the approaches that allow the designer to run his/her applications on an eventually consistent key-value store and handle data anomalies if they occur during the computation. To that end, we investigate two options: (1) Using detect-rollback approach, and (2) Using stabilization approach. In the first option, the designer identifies a correctness predicate, say Φ, and continues to run the application as if it was running on sequential consistency, as our system monitors Φ. If Φ is violated (because the underlying key-value store provides eventual consistency), the system rolls

back to a state where Φ holds and the computation is resumed from there. In the second option, the data anomalies are treated as state perturbations and handled by the convergence property of stabilizing algorithms.

We choose LinkedIn's Voldemort key-value store as the example key-value store for our study. We run experiments with several graph-based applications on Amazon AWS platform to evaluate the benefits of the two approaches. From the experiment results, we observe that overall, both approaches provide benefits to the applications when compared to running the applications on sequential consistency. However, stabilization provides higher benefits, especially in the aggressive stabilization mode which trades more perturbations for no locking overhead. The results suggest that while there is some cost associated with making an algorithm stabilizing, there may be a substantial benefit in revising an existing algorithm for the problem at hand to make it stabilizing and reduce the overall runtime under eventual consistency.

There are several directions of extension. For the detect-rollback approach, we are working to develop a more general rollback mechanism for the applications and improve the efficiency and accuracy of the monitors. For the stabilization approach, we are working to develop an analytical model for the benefits of eventual consistency in stabilizing programs. Our current work focuses on silent stabilization and we plan to extend our approach to other variations of stabilization.

To my family, whose love and support has made this dissertation possible.

ACKNOWLEDGEMENTS

I want to express my deep gratitude to my advisor, Professor Sandeep S. Kulkarni, for his kind help and support throughout my studies. Professor Kulkarni is always available to answer questions and explain the problems for me. He has spent countless hours to help me revise the manuscripts and comment my presentations. He introduces me to the opportunities at summer schools and conferences to extend my knowledge. I am very fortunate to have a knowledgeable, patient, and supportive advisor like him.

I am sincerely thankful to Professor Philip McKinley, Professor Eric Torng, and Professor Subir Biswas for having arranged time from their full schedules to serve in my Ph.D. guidance committee and provided valuable suggestions to improve my dissertation.

I am fortunate to have the opportunity to collaborate or be a lab mate of wonderful people: Professor Murat Demirbas, late Professor Ajoy K. Datta, Dr. Reza Hajisheikhi, Dr. Ling Zhu, Dr. Mohammad Roohitavaf, Dr. Vidhya Tekken Valapil, Mr. Sorrachai Yingchareonthawornchai, Dr. Aleksey Charapko. I am thankful for their kind advice, teaching, and help.

I also want to thank the office staff in the Department of Computer Science and Engineering, the Graduate School, the Office for International Students and Scholars, and other graduate students, friends at Michigan State University for their understanding and support. Their help has made this journey easier and full of happy memories.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xii

LIST OF ALGORITHMS xv

CHAPTER 1 INTRODUCTION 1
- 1.1 The Trade-off between Consistency and Performance in Distributed Key-value Stores. 1
- 1.2 Problem Statement 3
- 1.3 Approaches 5
- 1.4 Contributions 7
- 1.5 Outline of the Dissertation 12
- 1.6 Nomenclature 13

CHAPTER 2 PRELIMINARIES 18
- 2.1 Predicate Detection in Distributed Systems 18
  - 2.1.1 System Model 18
  - 2.1.2 Causality in Distributed Systems 18
  - 2.1.3 Vector Clocks 20
  - 2.1.4 Hybrid Vector Clocks 21
  - 2.1.5 A Basic Framework of Predicate Detection 22
  - 2.1.6 Linear Predicate and Detection Algorithm 25
  - 2.1.7 Semilinear Predicate and Detection Algorithm 28
- 2.2 Key-Value Store 29
  - 2.2.1 General Architecture of a Key-Value Store 29
  - 2.2.2 Voldemort Key-Value Store 30
  - 2.2.3 The Performance Difference between Eventual and Sequential Consistency in Voldemort Key-Value Store 30
- 2.3 Distributed Programs 33
  - 2.3.1 Traditional/Active-Node Model 34
  - 2.3.2 Passive-Node Model 34
  - 2.3.3 Similarity between Active-Node and Passive-Node Model 35
  - 2.3.4 Executing a Node Action by Client 36
  - 2.3.5 Stabilization 37
- 2.4 Consistency Violating Faults (cvf) 38

CHAPTER 3 DETECT-ROLLBACK APPROACH 43
- 3.1 Predicate Detection Module 43
  - 3.1.1 Overall Architecture 43
  - 3.1.2 Local Predicate Detector 46
  - 3.1.3 Implementation of the Monitors. 47

3.2 Rollback from Violations . . . 51
3.2.1 Rollback Mechanism . . . 51
3.2.2 Dealing with Potential of Livelocks . . . 54
3.3 Evaluation Results . . . 54
3.3.1 Experimental Setup . . . 54
3.3.2 Analysis of Throughput . . . 60
3.3.3 Analysis of System and Application Factors . . . 62
3.3.4 Analysis of Violations and Detection Latency . . . 65
3.3.5 Evaluating Strategies for Handling Livelocks . . . 66
3.3.6 Analysis of Applications . . . 67
3.4 Summary . . . 71

CHAPTER 4 STABILIZATION APPROACH . . . 73
4.1 Expected Properties of cvf. . . . 73
4.2 Termination Detection Algorithms. . . . 74
4.3 Experimental Evaluation of Benefits of Stabilization in Key-Value Stores . . . 75
4.3.1 Experiment Setup . . . 75
4.3.2 Experiment Results . . . 76
4.4 Discussion and Extensions . . . 80
4.4.1 Benefits with Active Stabilization . . . 81
4.4.2 Benefits with Contained Active Stabilization . . . 83
4.4.3 Benefits with Fault-Containment stabilization. . . . 85
4.4.4 Other Traditional Models of Computation . . . 86
4.4.5 Dealing with Non-Silent Algorithms . . . 86
4.4.6 Non-stabilizing Algorithms and cvf . . . 87
4.5 Summary . . . 88

CHAPTER 5 STABILIZATION VERSUS DETECT-ROLLBACK . . . 90
5.1 Experiment Setup . . . 90
5.1.1 System Configuration . . . 90
5.1.2 Client Execution Modes. . . . 91
5.1.3 Case Study Problems . . . 92
5.1.4 Input Graphs . . . 95
5.1.5 Workload Partitioning Schemes. . . . 95
5.1.6 Performance Metrics . . . 96
5.2 Benefits of Stabilization versus Rollback: Comparison and Analysis . . . 96
5.2.1 Stabilization vs. Rollback: Comparison and Analysis . . . 96
5.2.2 Improving the Convergence Time of Stabilization Approach . . . 102
5.2.3 Experiments on Amazon AWS . . . 106
5.2.4 Scalability Analysis . . . 106
5.2.5 Key Observation . . . 108
5.3 Analysis of Results and Their Implications in the Design . . . 108
5.3.1 Insight into Comparison of Stabilization versus Rollback . . . 108
5.3.2 Results with Non-Stabilizing Algorithm . . . 109
5.4 Summary . . . 110

CHAPTER 6 FUTURE WORK 112
6.1 Improving The Detect-Rollback Approach 112
6.2 Improving The Stabilization Approach 113
6.3 Other Possibilities of Future Work 115
6.3.1 Characteristics of Monitoring Errors 115

CHAPTER 7 RELATED WORK 117
7.1 Distributed Data Processing 117
7.2 Consistency in Distributed Data Stores 117
7.3 Predicate Detection in Distributed Systems 118
7.4 Distributed Snapshots and Reset 119
7.5 Monitoring Large-scale Web-services and Cloud Computing Systems. 120
7.6 Self Stabilization 121
7.7 Summary 121

CHAPTER 8 CONCLUSION 122

APPENDIX 125

BIBLIOGRAPHY 128

LIST OF TABLES

Table 1.1: List of Notations and Their Meanings. 14

Table 2.1: Examples of Possibly-Φ and Definitely-Φ for the computation in Figure 2.3 . . . 26

Table 3.1: Machine configuration in local lab experiments 55

Table 3.2: Setup of consistency models with N (replication factor), R (required reads), and W (required writes) 56

Table 3.3: Overhead (oh) and benefit of monitors in local lab network. For *Conjunctive* and *Weather Monitoring*, PUT percentage is 50%. 64

Table 3.4: Response time in 20,647 conjunctive predicate violations 66

Table 4.1: Benefit of Eventual Consistency in the Presence of $cvfs$ over Sequential Consistency. 15 Clients. With Local Mutual Exclusion. Convergence Time Unit: second. 77

Table 4.2: Revisiting Local Mutual Exclusion (lme): Treating Violations as $cvfs$. no-lme means without local mutual exclusion. lme means with local mutual exclusion. Number of clients is 15. Convergence Time Unit: second. 78

Table 4.3: Effect of Increased Concurrency on the Benefit of Eventual Consistency in the Presence of $cvfs$ over Sequential Consistency. 10,000-nodes *random-match* graph. Convergence Time Unit: second 79

Table 4.4: AWS Experiments. Benefit of Eventual Consistency in the Presence of $cvfs$ over Sequential Consistency. 15 Clients. Convergence Time Unit: second. 80

Table 5.1: Configurations of machines used in the experiments 91

Table 5.2: Four client execution modes 92

Table 5.3: Stabilization vs. Rollback. Graphs are partitioned in normal scheme. Network latency was 20 *ms*. *SEQ* is baseline for comparison. Rows 7-10 are convergence time benefits, shown in percentage increase or in speedup (e.g. $\times$5.2 means 5.2 times faster). 97

Table 5.4: Effect of random partitioning on stabilization and detect-rollback. Rows 2-5 are convergence time. Rows 6-8 are benefits, in percentage increase or in speedup (e.g. $\times$3 means 3 times faster). Network latency was 20 *ms* 101

Table 5.5: Comparison between the normal and random partitioning schemes of a planar graph. For each property, the average (AVG) and standard deviation (STDEV) among the partitions are calculated. 101

Table 5.6: Impact of Metis partitioning scheme. Latency was 20 *ms*. 102

Table 5.7: Effectiveness of the random coloring and the optimization for stabilization approach in the arbitrary graph coloring problem (*COLOR*). Convergence time is measured in seconds. Normal partition. Latency = 20 *ms* 104

Table 5.8: Impact of network latency. Rows 4-6 are convergence time (in seconds). Rows 7-8 are the benefits, shown in percentage increase or in speedup (e.g. ×4.3 means 4.3 times faster). 105

Table 5.9: Experiment results on Amazon AWS network 106

Table 5.10: (AWS) Performance of *COLOR* for different graph sizes. Normal partitioning . . 107

Table 5.11: Computation time (in seconds) of Stabilizing and Non-Stabilizing algorithms for graph coloring. The average of node degree (d) varies between 2 and 10. The baseline for calculating benefit is *SEQ* 110

LIST OF FIGURES

Figure 1.1: Illustration of two consistency models for key-value store. Original $X = 0$. In sequential consistency (Figure 1.1a), the *PUT* (write) operation would not succeed until the network condition is recovered because it requires confirmations from all replicas. Hence, clients always have a consistent view of the data under sequential consistency. In eventual consistency (Figure 1.1b), the *PUT* operation succeeded but clients observed different values of X. 3

Figure 1.2: Illustration cvfs: data anomalies in eventual consistency lead to incorrect computation results. Figure 1.2a: two clients use Peterson algorithm for mutually exclusive access to the critical section. However, the mutual exclusion requirement is violated if the key-value store is eventually consistent. Operations related to variable `turn` are not shown since the conditions of the while loop become false due to variables `x_1` and `x_2`. Figure 1.2b: Clients execute incorrect actions in graph coloring computation due to violation of mutual exclusion. On the left is the original color and on the right is the new coloring after client 1 and client 2 executes actions on nodes B and C, respectively. The new colors resulted from those actions are still not a valid coloring. 6

Figure 1.3: The detect-rollback approach: when the predicate of interest is violated, system state is restored to the most recent consistent snapshot and the computation resumes from there. 8

Figure 2.1: (This figure is based on [1]). The lattice of distributed computation history (right) is constructed from a particular execution (left) and causality. $x + y > 15$ is a *definitely* predicate since it is met by state S_{31} and every computation passes through S_{31}. $x + y = 12$ is a *possibly* predicate since it is met by only S_{21} and there are computation paths not going through S_{21}. 23

Figure 2.2: Reduction of a SAT instance to a GLOB instance. 24

Figure 2.3: An example of a distributed computation 26

Figure 2.4: Illustration of semi-forbidden state. 28

Figure 2.5: Illustration of locks. To update node 6, a client has to obtain these locks in following order: L_1_6, L_5_6, L_6_9 37

Figure 2.6: Illustration of cvf in Voldemort. Clients run on eventual consistency R1W1 . . 40

Figure 2.7: A computation in the presence of cvf 42

Figure 3.1: Architecture of predicate detection module 44

Figure 3.2: XML specification for $\neg\mathcal{P} \equiv (x_1 = 1 \wedge y_1 = 1) \vee z_2 = 1$ 45

Figure 3.3: Illustration of candidates sent from a server to monitors corresponding to three conjunctive predicates. If the predicate is semilinear, the candidate is always sent upon a PUT request of relevant variables. 47

Figure 3.4: Illustration of causality relation under HVC interval perspective 48

Figure 3.5: Two client tasks involved in a violation. Since detection latency is much smaller than the *Read* phase time, violation will be notified within *Read* phase of the current task of at least one client. 52

Figure 3.6: Simulating network delay using proxies. The proxies virtually partition our local lab network into three regions 55

Figure 3.7: Illustration of result stabilization. The *Social Media Analysis* application is run three times on Amazon AWS with monitoring enabled. Number of servers $(N) = 3$. Number of clients per server $(C/N) = 5$. Aggregated throughput measured by *Social Media Analysis* application in three different runs and their average is shown. This average is used to represent the stable value of the application throughput. 60

Figure 3.8: (AWS) *Social Media Analysis* application, 3 servers, 15 clients. The benefit of eventual consistency with monitors vs. sequential consistency without monitors (throughput improvement compared to R1W3 and R2W2 is 57% and 78%, respectively), and the overhead of running monitors on each consistency setting (the overhead is less than 2%). 61

Figure 3.9: Benefit and overhead of monitors in *Weather Monitoring* application. Percentage of PUT requests is 25% and 50% Number of servers =5. Number of clients = 10. Machines are on the AWS North Virginia region but in different availability zones. 62

Figure 3.10: Effectiveness of livelock handling mechanisms. Number of servers=3, number of clients=30. We observed that adaptive mechanism worked best for *Social Media Analysis* (Figure 3.10a), and backoff mechanism worked best for *Weather Monitoring* (Figure 3.10b). 67

Figure 3.11: The benefit and overhead of Eventual consistency+Rollback vs. Sequential consistency in *Weather Monitoring* application. The inset figure within Figure 3.11b is a close-up view showing the impact of rollback. The larger points near the end of each data sequence are where we choose the representative values for the data sequences. 69

Figure 3.12: Comparing the completion time of Sequential Consistency (R1W3) vs. Eventual Consistency with rollback and adaptive consistency (R1W1+adaptive) in *Social Media Analysis* application. On a power-law clustering graph, before 90% of the nodes are processed, R1W1+adaptive progresses about 18% faster than R1W3. Overall, R1W1+adaptive is 9.5% faster than R1W3. On a regular random graph, the benefit before 90% of the nodes are processed is 26% and the overall benefit is 20.8%. 70

Figure 4.1: Convergence of maximal matching 79

Figure 4.2: Convergence of maximal matching in the experiments deployed on Amazon EC2 instances. Note that this convergence pattern is similar to the convergence pattern in Figure 4.1 except that the convergence in Amazon EC2 experiments converges slower. This is because the delay in Amazon AWS network is longer. 81

Figure 5.1: Illustration of solving a traffic phasing problem using graph coloring. Each color in the lower right graph corresponds to one time slot of green light. 93

Figure 5.2: Measurement of client throughput (*ops* – operations per seconds) of *MAXMATCH* with different input graphs. Normal partitioning. Latency was 20 *ms*. 99

Figure 5.3: The convergence pattern of different execution modes in *COLOR*. Normal partitioning. Latency was 20 *ms*. 104

LIST OF ALGORITHMS

Algorithm 1 Linear predicate detection algorithm adapted from [2] 27

Algorithm 2 Semilinear predicate monitor algorithm adapted from [3] 29

Algorithm 3 Rollback algorithm at a client 53

CHAPTER 1

INTRODUCTION

1.1 The Trade-off between Consistency and Performance in Distributed Key-value Stores.

Distributed key-value stores [4–10] have gained increasing popularity due to their scalability and simple data model. Clients (users) view a distributed key-value store as a single table with two fields: a unique *key* field for storing the variable names, and a *value* field for storing the associated values of the variables. A machine storing that table is called a *replica* (or *server*). Clients' access to the key-value store is performed by issuing two operations, *PUT* (write) and *GET* (read), to the replicas.

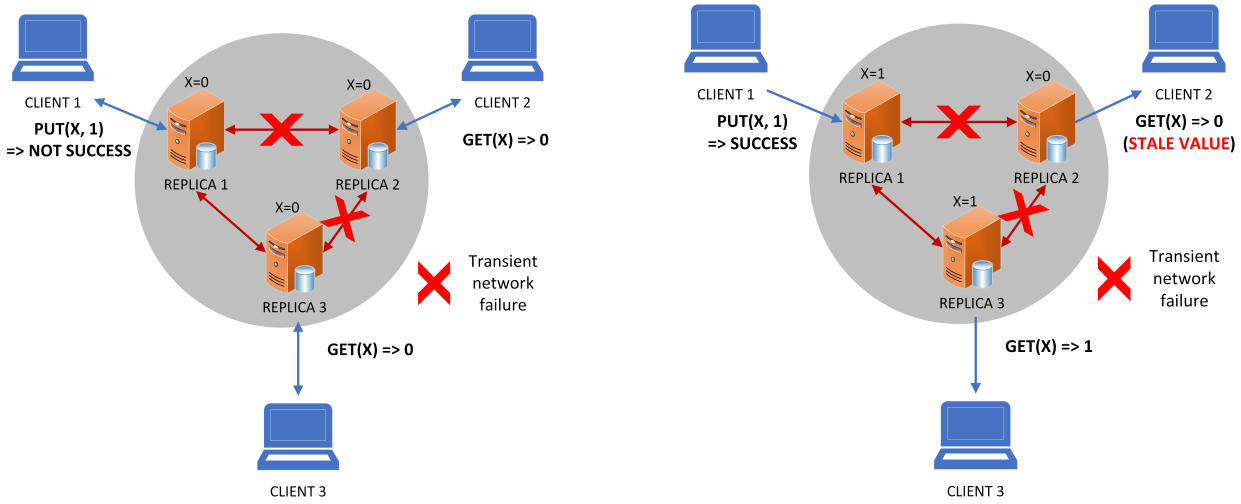
The *clients* are distributed programs that coordinate to perform some computation task (e.g. find a valid coloring for a graph). During their execution, clients use the key-value store to retrieve data and update computation results (i.e. permanent data are not kept locally at the clients). For the client program to be correct, it is required that the values read by the clients are up-to-date. This requirement can be divided into two smaller requirements: (1) the client actions are atomic [11] (e.g. a client should not read a value that is being updated by another client), and (2) the data store is sequentially consistent [12], i.e. responses from different replicas are identical so that the clients have the illusion that they are interacting with a single replica.

The first requirement can be satisfied if the clients employ some mutual exclusion mechanism such as locking [13] when they access data entries. The second requirement is trivially satisfied if the key-value store consists of one replica. However, single-replica key-value store is not used in practice since the replica will become a performance bottle-neck as well as a single point of failure. To provide higher availability and fault tolerance, the key-value table is replicated across multiple replicas located at regions geographically far enough (a failure in a region does not affect other regions). The key-value store also employs a consistency model (protocol) to keep the data at

replicas synchronized. To meet the second requirement aforementioned, the model employed should be the sequential consistency model. Sequential consistency is *more natural* for programmers to write programs as it masks the complexity of replication and ensures that different clients always observe the same value for the same key. However, when there is a transient failure, in order to fulfill that contract, the sequential consistency model would block client operations until it is safe to proceed (cf. Figure 1.1a). This restriction impedes the performance (throughput and latency of operations) of the key-value store.

Another consistency model is eventual consistency which is a best-effort approach. When a client reads the value of a key, each replica returns the most recent value that it knows (even when that value differs from the values stored at other replicas). When a client updates a key, a confirmation from one replica is sufficient for the *PUT* operation to be considered successful. (Confirmations from multiple replicas are great but not required, and the responsive replica is usually located in the same geographical region as the client). Since there is no blocking, the performance of a key-value store under eventual consistency is higher than under sequential consistency. However, when there is a transient network failure, different replicas could store different values for the same key under eventual consistency. If this happens, clients may observe different values for the same key (cf. Figure 1.1b) and they could perform undesirable actions. Although such *data anomalies* are not frequent [4] and they are expected to be resolved eventually when the failure stops and new updates override the old values, they affect the correctness of the computation and compromise subsequent execution.

The advantages and disadvantages of the sequential and eventual consistency models reflect the inevitable trade-off between consistency and performance in a key-value store. Recall that due to the CAP theorem [14, 15], it is impossible for a distributed key-value store to simultaneously achieve three properties: (C) sequential consistency, (A) availability, i.e. a client request is always satisfied within a provisioned time, and (P) partition tolerance, i.e. the key-value store is still operational despite the presence of network failures. Since failures are a norm in distributed systems and the capability to maintain the service at an acceptable level in the presence of failures



(a) Sequential consistency

(b) Eventual consistency

Figure 1.1: Illustration of two consistency models for key-value store. Original $X = 0$. In sequential consistency (Figure 1.1a), the *PUT* (write) operation would not succeed until the network condition is recovered because it requires confirmations from all replicas. Hence, clients always have a consistent view of the data under sequential consistency. In eventual consistency (Figure 1.1b), the *PUT* operation succeeded but clients observed different values of X.

is a critical dependability and business requirement of any system, the partition tolerance property is a necessity. Consequently, the trade-off between consistency and availability (performance) is inevitable. The best trade-off decisions between consistency and performance often depend on the specific applications carried out by the clients and there is an array of consistency models designed for different needs [16, 17]. This dissertation focuses on sequential consistency and eventual consistency as they are the consistency models available in most distributed key-value stores [4, 5, 10, 18–24]. In this dissertation, we choose LinkedIn's Voldemort [5] key-value store as the example of key-value store for studying and evaluation since the project is open-source and supports tunable consistency.

1.2 Problem Statement

The problem we are interested in this study is as follows:

A designer has to solve a distributed computation problem on a key-value store. The key-value store only provides sequential consistency and eventual consistency, and the key-value store has better performance under eventual consistency. What should the designer do to make use of this advantage of eventual consistency?.

In order to use eventual consistency, the designer has to address the problem associated with it: data anomalies could lead to incorrect computation results. As an illustration of this problem, consider a distributed computation that relies on a key-value store to arrange exclusive access to a critical resource for the clients. If the key-value store employs sequential consistency and the clients use Peterson's algorithm [13] then the mutual exclusion is guaranteed but the performance is slow. If eventual consistency is adopted then the mutual exclusion is violated (cf. Figure 1.2a). The reason for this violation is that when there is a transient network failure, client requests are still served by the local replicas (servers) and the clients will observe different values for the variables related to Peterson algorithm (namely `x_1`, `x_2`, and `turn`). Both clients think that the status of the lock is available and they proceed to the critical section simultaneously. We say that two clients *conflict* when the time intervals during which the clients want to access the same critical section overlap.

For the sake of discussion, suppose that the computation task carried out by the clients is the graph coloring problem [25] in which the clients have to assign colors to every graph node (vertex) so that neighboring nodes have different colors. Each client is assigned to work on a partition of the input graph (a subset of graph nodes). Assume client 1 and client 2 are working on two neighboring nodes B and C, respectively. To ensure action atomicity, a lock is imposed on the edge (B, C) so that client 1 (respectively client 2) could not process node B (respectively node C) if it has not obtained the lock corresponding to edge (B, C). As illustrated before, this lock could be (incorrectly) obtained by both clients under eventual consistency. Each client then proceeds to perform the action on its node, i.e. it reads the colors of neighboring nodes and chooses a different color for its node. Suppose the initial color of every node is 0, both clients choose color 1 as the new color for both B and C. However, these client actions are incorrect since B and C are neighbors

so they should have different colors (cf. Figure 1.2b). The reason for these incorrect actions is that client 1 (respectively client 2) reads the color of node C (respectively node B) while that color is being updated by client 2 (respectively client 1). If the clients use locks and the key-value store is sequentially consistent, the resultant coloring is always valid since at most one client is allowed to proceed and update the color of its node at any given time.

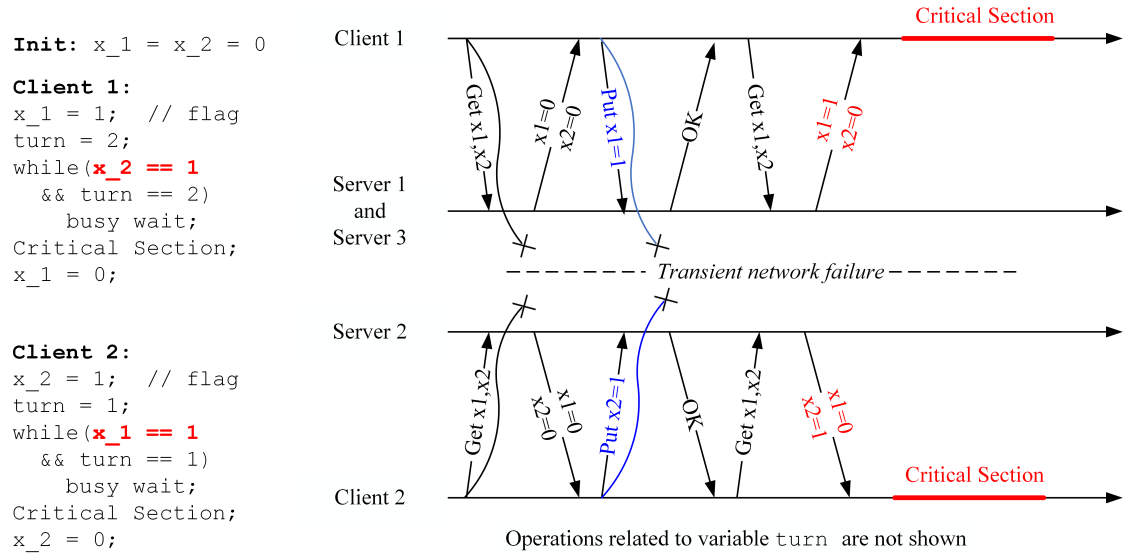
We denote such erroneous client actions as *consistency violating faults* (cvfs) because the consequences of those actions are similar to consequences of faulty program transitions and the causes of those actions are due to data anomalies in eventual consistency. We will formally define cvfs in Section 2.4.

1.3 Approaches

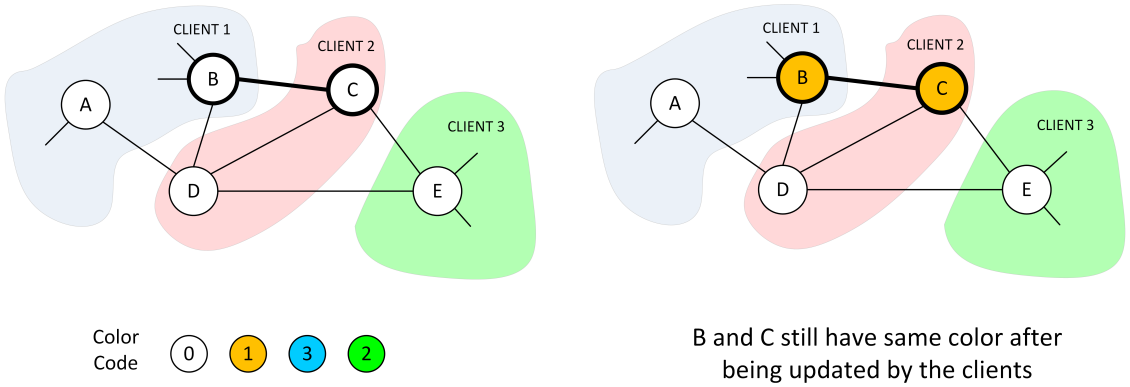
To address the problem caused by cvfs, the designer has two options: either prevent cvfs by employing sequential consistency or let cvfs occur in eventual consistency and handle them. The first option retards the performance and is not in accordance with the goal of the problem we have stated before. For the second option, there are several approaches:

1. Develop a brand new algorithm that works under eventual consistency, or
2. Use stabilization to handle cvfs, or
3. Run an existing algorithm (available for sequential consistency) on eventual consistency by pretending that the underlying system satisfies sequential consistency but monitor the execution to detect violations of the mutual exclusion requirement and perform corrective actions upon a violation is reported.

In case of the first approach, we potentially need to develop a new algorithm for each computation task at hand. In case of the second option, data anomalies and the consequences of cvfs are treated as state perturbations and a stabilizing algorithm is already designed to handle the issue. In case of the third option, the corrective actions may include rolling back the system to an earlier state if a violation is found. While rollback in general distributed systems is a challenging task, existing



(a) Violation of mutual exclusion



(b) Incorrect client actions in graph coloring

Figure 1.2: Illustration $cvfs$: data anomalies in eventual consistency lead to incorrect computation results. Figure 1.2a: two clients use Peterson algorithm for mutually exclusive access to the critical section. However, the mutual exclusion requirement is violated if the key-value store is eventually consistent. Operations related to variable `turn` are not shown since the conditions of the while loop become false due to variables `x_1` and `x_2`. Figure 1.2b: Clients execute incorrect actions in graph coloring computation due to violation of mutual exclusion. On the left is the original color and on the right is the new coloring after client 1 and client 2 executes actions on nodes B and C, respectively. The new colors resulted from those actions are still not a valid coloring.

approaches have provided rollback mechanisms for key-value stores with low overhead such as [26]. Moreover, it is possible to develop efficient application-specific rollback algorithms by exploiting the properties of the applications. This study considers the approaches that allow users to use existing algorithms with minimal modification and run them on eventual consistency. In particular,

the second and third approaches aforementioned (stabilization and detect-rollback) are our main interest. We also note that although we focus on eventual consistency in this study, these two approaches are applicable for any consistency model weaker than sequential consistency as well.

While this work could be potentially useful for distributed computation problems in general, the focused applications of this work are distributed computation problems on graphs. In these problems, the state of each graph node depends on its neighbors. Each client is assigned a partition of the graph (a subset of graph nodes) in such a way that the workload is roughly evenly distributed among the clients. Since the state of a node depends on its neighbors, the clients need to coordinate to avoid executing actions on two neighboring nodes simultaneously. Otherwise, they may read *inconsistent data* (data being updated by some client and other clients should not read this data item until the update is complete) and their computation results would be incorrect as illustrated above in the graph coloring problem (cf. Figure 1.2b). This coordination requirement manifests in many other graph computation problems as well such as spanning trees [27, 28], leader election [29, 30], matching [31, 32], dominating set [33, 34], independent set [35, 36], clustering [37–39].

1.4 Contributions

This dissertation focuses on two approaches that handle cvfs which occur during the execution of distributed graph computations on eventually consistent key-value stores, namely the stabilization and detect-rollback approaches. Specifically, it evaluates the benefits of these two approaches when compared to the baseline of using sequential consistency (i.e. the option of preventing cvfs). In this direction, the dissertation has the following contributions:

(1) **Detect-rollback approach**: we propose a detect-rollback framework to handle cvfs during the execution of distributed computations on eventually consistent key-value stores. Specifically, the designer identifies a correctness predicate, say Φ, and runs the distributed computation (using the algorithms designed for sequential consistency) on eventual consistency. At the same time, a monitoring module will monitor for violation of Φ (as a consequence of cvfs) during the execution. If Φ is violated ($\neg\Phi$ is true), the systems will be rolled back to

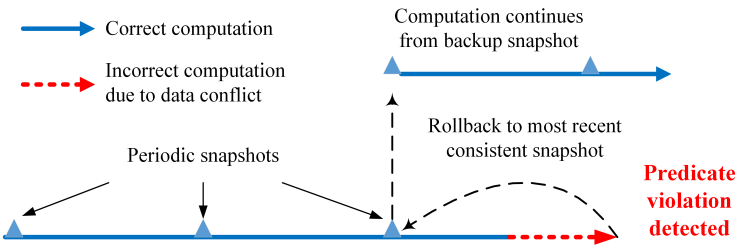


Figure 1.3: The detect-rollback approach: when the predicate of interest is violated, system state is restored to the most recent consistent snapshot and the computation resumes from there.

an earlier correct state from where subsequent execution is resumed.

To monitor Φ, we design and develop a prototype of the monitoring module for the Voldemort key-value store [5]. Our monitors adapt the algorithms by Garg and Chase [2,3] for detection of linear and semi-linear predicates on key-value stores.

For the rollback task, one possibility is restoring the system to a recent snapshot taken by lightweight snapshot tools such as Retroscope [26] (cf. Figure 1.3). This is the most general option in the sense that it can be used for any application. However, it is potentially expensive [1]. Another possibility of rollback is analyzing and exploiting the properties of violations in specific applications. In this regard, we propose an efficient rollback algorithm specifically designed for distributed graph computations.

We use several graph-based applications that are motivated by the task of *Social Media Analysis* and *Weather Monitoring* as our test cases (the details of these applications are described in Section 3.3.1). We run these test cases on Amazon AWS platform and on our local lab network (where we can control the network latency) to quantify the usefulness of the proposed detect-rollback approach (using the rollback algorithm specific for graph computations). From the experimental results, we observe the following benefits of the detect-rollback approach:

- **Throughput improvement**: We observe that the Voldemort key-value store achieves significantly higher performance under eventual consistency (even with monitors run-

[1] We are working on the general rollback mechanism with Retroscope

ning concurrently with the applications) than under sequential consistency. Specifically, the client throughput is improved by 50% – 80% for *Social Media Analysis* applications and by 20% – 50% for *Weather Monitoring* applications. We note that this throughput improvement is not necessarily the improvement in the overall computation time since some requests such as checking the status of the locks are counted in the throughput measurement but do not contribute to the progress of the computation.

- **Efficient detection**: The monitors could quickly detect violations of linear and semi-linear predicates even when there are as many as thousands of predicates being monitored simultaneously. In particular, more than 99.9% of violations were detected within 50 milliseconds for experiments on Amazon AWS regional network (all machines in the same AWS region), and within 3 seconds on the global network (machines in different AWS regions).
- **Low monitoring overhead**: We evaluate the overhead of the monitoring module to see how it might affect the computation when being used for the purpose of debugging or runtime monitoring. We observe that when the monitors were used with sequential consistency, the overhead was at most 8%. And, for eventual consistency, the overhead was less than 4%.
- **Improvement in the final benefits**: We observe that the final benefits (i.e. the benefits observed in the final progress of the applications) vary depending on the properties of applications. Specifically, for non-terminating applications such as *Weather Monitoring*, the application progress is 45% – 47% faster on detect-rollback than on sequential consistency. For terminating applications such as *Social Media Analysis*, the benefit is 10% – 20%.

  One of the reasons for the reduced benefit in terminating applications is that during the final phase of a terminating application, there are few tasks to be processed, thus the chance of client conflicts and recurring violations is increased during this phase. In fact, if the application keeps using eventual consistency, the computation may *stall* due to

repeated rollbacks (livelocks). We use strategies such as random backoff and adaptive consistency to overcome livelocks. We also observe that terminating applications using detect-rollback approach progressed 16%–28% faster than using sequential consistency during the first phase of the computation (the first phase is when 90% of the work is done, and the final phase is when 10% of the remaining work is done), and 10%–20% faster overall (because it has to switch from eventual consistency to sequential consistency during the final phase of the execution).

(2) **Stabilization approach**: Unlike the detect-rollback approach, the stabilization approach does not need additional mechanisms to handle cvfs except that the existing algorithm (for sequential consistency) is stabilizing [2]. We evaluate the difference in the performance when running a stabilizing algorithm on eventual consistency versus on sequential consistency. We choose the stabilizing algorithm for maximal matching by Manne et al. [40] as the case study and find that the convergence time of maximal matching is 1.2–1.8 times faster on eventual consistency than on sequential consistency. Especially, if we allow the algorithm to run on eventual consistency without a mutual exclusion mechanism and treat violations of mutual exclusion as additional cvf violations (we call this execution mode *aggressive stabilization*), the speedup factor is 7–12 times.

(3) **Stabilization vs. Detect-rollback**: We compare the benefits of the two approaches. Clearly, if the underlying program is not stabilizing then we must rely on the detect-rollback approach. Hence, we focus on stabilizing programs where both approaches are applicable. Specifically, we consider three stabilizing graph computation problems/applications: planar graph coloring, arbitrary graph coloring, and maximal matching. From the experimental results, we obtain the following observations:

- **Overall comparison**: Stabilization approach provides higher benefits than the detect-

[2] In this dissertation, the terms self-stabilizing and stabilizing are interchangeable, and so are self-stabilization and stabilization.

rollback approach in the three graph computation problems above. Using sequential consistency as the base-line for comparison, stabilization improves the convergence time of the programs by 25 % to 35 %, whereas the detect-rollback approach improves the convergence time by 30 % in the best case and potentially causes performance to suffer. Notably, the convergence time of the programs speeds up by 2 to 15 times with aggressive stabilization.

The aggressive stabilization execution mode eliminates the locking overhead at the cost of extra cvfs. The boosted speedup of aggressive stabilization suggests that the stabilization cost for the extra cvfs is outweighed by the benefits of no locking overhead. This observation is also compatible with our analysis in which we observe that a sizeable proportion of the client computation time is spent on obtaining locks to ensure the action atomicity requirement is satisfied (clients do not update the states of neighboring nodes simultaneously).

- **Analysis of the impact of cvfs**: We analyze the cvfs caused by the absence of locks and find that many of those cvfs resolve favorably by themselves (they do not result in erroneous computation), thus the actual stabilization cost is lower than what we have thought. In contrast, the removal of locks would require the detect-rollback approach to utilize more complicated mechanisms to detect atomicity violation instances. The overhead of such a mechanism is expensive, thus we do not consider the aggressive detect-rollback approach in this dissertation.
- **Mitigating cvfs in aggressive stabilization** Although being more beneficial, aggressive stabilization could suffer from some cvfs that prevent the programs to converge. We propose some heuristics to improve the performance of aggressive stabilization in such cases.
- **Impact of other factors**: The performance of detect-rollback and stabilization is analyzed under different dimensions such as types of case study problems, characteristics of input graphs, partitioning schemes, and network latency. We observe that for most of

these factors, the impact of a factor on stabilization and on detect-rollback is different.

- **Scalability**: When tested on large-scale real-world graphs, the stabilization approach, especially aggressive stabilization scales very well. This scalability is not observed in sequential consistency and in the detect-rollback approach.
- **A comparison with non-stabilizing algorithm**: A natural question could be that what options should we choose if both stabilizing and non-stabilizing algorithms are available? We note that in general, when the algorithms are different, it is hard to fairly compare the two approaches since the performance is affected not only by the algorithm itself but also by other factors such as optimization and implementation techniques. However, if the algorithms are closely similar, the comparison may be useful. In particular, we compare the stabilizing and non-stabilizing algorithms for graph coloring since the algorithms are fairly similar. A key observation we obtain from the experimental results is that the stabilizing algorithm is less efficient than the non-stabilizing counterpart on sequential consistency. However, it is the overall winner when used with eventual consistency, as it can benefit from tolerating cvfs. We also note that more studies are needed to obtain more insights and more comprehensive comparison.

These observations imply that while there is some cost associated with making an algorithm stabilizing, there may be a substantial benefit in revising an existing algorithm for the problem at hand to make it stabilizing and reduce the overall runtime under eventual consistency.

1.5 Outline of the Dissertation

The remaining of this dissertation consists of seven chapters and one appendix as follows:

- Chapter 2 provides the definitions of concepts and notations used in this dissertation. We reiterate the notion of causality and the general framework for predicate detection. Then we describe two classes of predicate that are used in this dissertation, linear and semilinear

predicates, and their detect algorithms. Then we review the general architecture of a key-value store and the specifics in the Voldemort key-value store. We also present the formal definition of distributed programs, the traditional active-node model and the new passive-node model, the notion of self-stabilization, and define the notion of consistency violating faults ($cvfs$).

- Chapter 3 investigates the benefits of the detect-rollback approach. For the detect phase of the detect-rollback approach, we present our design and implementation of the monitors and evaluate the performance of the monitors. For the rollback phase, we propose an application-specific rollback algorithm as well as strategies for handling livelocks (re-occurring violations). We also present evaluation results of rollback. These results have been published in [41, 42].

- Chapter 4 investigates the benefits of self-stabilization approach. We present our evaluation with the maximal matching self-stabilization program and discuss extensions of this approach in other versions of stabilization. The results in this chapter have been published in [43].

- Chapter 5 compares the benefits of detect-rollback approach and stabilization approach. We present the evaluation results under different factors such as case study problems, input graphs, partitioning schemes, etc. The results in this chapter have been published in [44].

- Chapter 6 discusses some possible directions to extend our results.

- Chapter 7 reviews the literature related to this dissertation and identifies the contributions of this dissertation.

- Chapter 8 concludes this dissertation.

- In the Appendix, we list the publications on which this dissertation is based. We also provide access links to the relevant source code and experimental data set.

1.6 Nomenclature

In Table 1.1, we list some common notations that we use in this dissertation and their meanings.

Table 1.1: List of Notations and Their Meanings.

Notation	**Meaning**
General:	
n	Number of processes in a distributed system
P_i	The i^{th} process
ϵ	The bound for clock synchronization error
Φ or $\mathcal{P}$	Predicate of interest. We want to detection violations of Φ or $\mathcal{P}$
cvf	Consistency Violating Faults
COLOR	The problem of coloring an arbitrary/general graph
P-COLOR	The problem of coloring a planar graph
MAX-MATCH	The problem of finding a maximal matching in a general graph
AWS	Amazon Web Service
Causality:	
e_{ij}	The j^{th} local event on process P_i
$e \rightarrow f$	Event e happens before event f
$e\ hb f$	Same as $e \rightarrow f$, event e happens before event f,
$e \| f$	Events e and f are concurrent
VC_i	Vector clock on process P_i
$VC_i[j]$	The j^{th} element of vector clock VC_i
$e.VC$	The vector clock associated with event e
$VC_1 < VC_2$	Vector clock VC_1 is smaller than vector clock VC_2
$VC_1 \| VC_2$	Vector clocks VC_1 and VC_2 are concurrent
HVC_i	Hybrid vector clock on process P_i
$HVC_i[j]$	The j^{th} element of hybrid vector clock HVC_i

Table 1.1 (cont'd)

Notation	**Meaning**
$HVC_1 < HVC_2$	Hybrid vector clock HVC_1 is smaller than hybrid vector clock HVC_2
$HVC_1 \| HVC_2$	Hybrid vector clocks HVC_1 and HVC_2 are concurrent
PT_i	The physical clock on process P_i
Predicate Detection:	
SAT	Satisfiablity problem
NP	Non-deterministic Polynomial time
GLOB	The problem of Global Predicate Detection
$X \leq Y$	The cut/state Y is reachable from the cut/state X
$sup(X)$	Set of supremal states of cut X
$succ(e)$	Successor of local state/event e
X^e	Advance of X along e
$final(X)$	X is final, i.e. execution stops at X
$P_\Phi(X, Y)$	Possibly-Φ
$D_\Phi(X, Y)$	Definitely-Φ
$forb_\Phi(s, X)$	s is forbidden state of cut X w.r.t. Φ
$sforb_\Phi(s, X)$	s is semi-forbidden state of cut X w.r.t. Φ
$eligible(X)$	Set of all eligible states of X
Key-value Store:	
Replica (or server)	A machine that stores the key-value store data
Client	A process/program that performs some distributed computation task. Its data (e.g. computation results) is not kept locally but kept in the key-value store.
PUT	Put/Write request that updates the value of a key
GET	Get/Read request that retrieves the value of a key

Table 1.1 (cont'd)

Notation	**Meaning**
GET_VERSION	Get version request that retrieves the version of a key
N	Replication factor, i.e. the number of copies for each data entry (key)
R	Required reads, i.e. the minimum number of responses needed for a *GET* (read) request to be successful.
W	Required writes, i.e. the minimum number of confirmations needed for a *PUT* (write) request to be successful.
$N3R1W3$	Example of consistency where $N = 3$, $R = 1$, $W = 3$
SEQ	Sequential consistency
EVE-S	Eventual consistency with stabilization
EVE-AS	Eventual consistency with aggressive stabilization (or aggressive stabilization for short)
Rollback	Eventual consistency with detect-rollback
Distributed Program:	
p	A distributed program
V_p	Set of nodes in program p
E_p	Set of edges in program p
var_p	Set of variables in program p
S_p	State space of program p
s_i	The i^{th} state of program p
ac_j	Set of actions at node j (of program p)
δ_{ac}	Set of transition of action ac
δ_p	Set of transition of program p
δ_j	Set of transition of node j
I	Invariant of program p

Table 1.1 (cont'd)

Notation	**Meaning**
adv_p	computation of program p in the presence of adversary
L_5_20	A lock corresponding to the edge between node 5 and node 50.

CHAPTER 2

PRELIMINARIES

This chapter defines the concepts used in this dissertation. Section 2.1 defines the model of distributed systems, the notion of causality, vector clocks, and the problem of predicate detection in distributed systems. We also discuss the linear and semilinear predicates and their detection algorithms as this dissertation focuses on these two classes of predicate. In Section 2.2, we review the general architecture of a key-value store and the specifics in the Voldemort key-value store. Section 2.3 presents the formal definition of distributed programs, the traditional active-node model and the new passive-node model, and the notion of self-stabilization. Finally, Section 2.4 define the notion of consistency violating faults (cvfs). The content of this chapter is mainly adapted from [1–3, 42, 43, 45–51].

2.1 Predicate Detection in Distributed Systems

2.1.1 System Model

A distributed system consists of n processes $P_1, P_2, ..., P_n$. Those processes do not have a shared memory and do not have a shared global clock. Each process operates according to its local algorithm and communicates with other processes using messages. During its execution, states of a process are changed by events. There are 3 types of events: sending message events, receiving message events, and internal computation events. The set of all events within a process P_i is denoted as $E_i = \{e_{i1}, e_{i2}, ...\}$. The set of all events that have happened during the computation of a distributed system is $E = \bigcup_i E_i$.

2.1.2 Causality in Distributed Systems

In the above asynchronous model of distributed systems, actions performed by processes are not determined by the real time but by the events. For example, a process only starts an internal

computation after it receives a particular message msg, no matter the real time when it receives msg is. A process only receives a message msg after some process sends msg. Therefore, the cause and effect relationship has an important role in connecting events in a distributed computation.

The cause and effect relationship between events is expressed by the notion of causality [45] (which is also known as happen-before relationship, or causal ordering). Specifically, an event e is said to happen before (or causally precede) an event f, denoted as $e \rightarrow f$ when one of the following conditions is met:

- e and f are events on the same process and e proceeds f. That is there exists P_i such that e and f are local events of P_i: $e = e_{ij}$ and $f = e_{ik}$, and $i < k$.
- e is the sending event of a message msg, and f is the receiving event of that message msg.
- There exists some event g such that $e \rightarrow g$ and $g \rightarrow f$.

We note that the causality defined above captures the *potential* cause and effect relationship which sometimes may not be the actual one. For example, events e and f are two message-receive events on the same process and e occurs before f. By definition $e \rightarrow f$, but in reality, the occurrence of e does not causally affect the occurrence of f. If we re-run the computation again, it is possible that f occurs before e.

If neither e causally precedes f nor f causally precedes e, then e and f are said to be concurrent, denoted as $e \| f$. Specifically,

$$e \| f \iff \neg(e \rightarrow f) \wedge \neg(f \rightarrow e)$$

Causality relationship is used to define consistent global states. A consistent global state is the state resulted from a consistent cut. A consistent cut is a set of events in a distributed computation such that if the cut includes an event e then it also includes all events causally preceding e. In other words, for a cut to be consistent, if it includes the effect then it also includes the causes. However, a consistent cut could include the causes without consequent effects. Formally, a cut C is consistent

when

$$e \in C \wedge f \rightarrow e \implies f \in C$$

Equivalently, a global snapshot is considered consistent if its constituent local states are mutually concurrent. Since a process state results from an event, if there are local states that are not mutually concurrent, there exists an event that is not included in the cut but one of its effects is included in the cut.

Causality relationship could be represented by different mechanisms such as vector clocks (or vector time) [46, 47], dependency vectors, and concurrent maps [52]. Among them, vector clock is the most popular mechanism.

2.1.3 Vector Clocks

Vector clocks, defined by Fidge and Mattern [46, 47], are designed for asynchronous distributed systems that make no assumption about underlying speed of processes or about message delivery. Each process P_i maintains a vector $VC_i[]$ of n integer numbers which represent the best information that P_i knows about the clocks of other processes. Initially, $VC_i[j] = 0 \ \forall i, j$. Vector clocks are updated according to the following rules:

- Upon the occurrence of a local event (internal computation, send message, receive message) at process P_i, the i^{th} element of VC_i is first incremented by 1, i.e. $VC_i[i] = VC_i[i] + 1$.
- Each message sent by process P_i will also include the latest vector clock VC_i.
- When a process P_i receives a message, it will update every component of its vector clock to the most recent known value:

  $$\forall j : \quad VC_i[j] = max(MVC[j], VC_i[j])$$

  s where MVC is the vector clock piggy-backed on the message by its sender.

.

Vector clocks can be compared using vector comparison method (suppose VC_1 and VC_2 are two vector clocks):

$$VC_1 < VC_2 \iff (\forall j : VC_1[j] \leq VC_2[j]) \wedge (\exists k : VC_1[k] < VC_2[k])$$
$$VC_1 \| VC_2 \iff \neg(VC_1 < VC_2) \wedge \neg(VC_2 < VC_1)$$

It can be proved that vector clocks exactly characterize causality relationship. Let $e.VC$ denote the vector clock associated with event e, then:

$$e \rightarrow f \iff e.VC < f.VC$$
$$e \| f \iff e.VC \| f.VC$$

Hence, in order to determine the causal relationship between events or states, we just compare their corresponding vector clocks.

2.1.4 Hybrid Vector Clocks

Hybrid vector clocks [48] are designed for systems where clocks of processes are synchronized within a given synchronization error (denoted as parameter ϵ in this dissertation). While the size of vector clocks is always n (the number of processes in the system), hybrid vector clocks have the potential to reduce the size to less than n. Basically, a hybrid vector clock is a vector clock with the optimization that clock elements that have not been updated for a long time will be automatically updated (because processes clocks are partially synchronized) and their representation can be compacted to reduce the space.

Every process maintains its own HVC. HVC at process P_i, denoted as HVC_i, is a vector with n elements such that $HVC_i[j]$ is the most recent information process P_i knows about the physical clock of process P_j. $HVC_i[i] = PT_i$, the physical time at process i. Other elements $HVC_i[j], j \neq i$ is learned through the communication between processes. When process P_i sends a message, it updates its HVC as follows: $HVC_i[i] = PT_i$, $HVC_i[j] = max(HVC_i[j], PT_i - \epsilon)$ for $j \neq i$. Then HVC_i is piggy-backed with the outgoing message. Upon reception of a message msg,

process P_i uses the piggy-backed hybrid vector clock HVC_{msg} to update its HVC: $HVC_i[i] = PT_i$, $HVC_i[j] = max(HVC_{msg}[j], PT_i - \epsilon)$ for $j \neq i$.

Hybrid vector clocks are vectors and can be compared as usual. Given two hybrid vector clock HVC_i and HVC_j, we says HVC_i is smaller than HVC_j, denoted as $HVC_i < HVC_j$, if and only if $HVC_i[k] \leq HVC_j[k] \forall k$ and $\exists l : HVC_i[l] < HVC_j[l]$. If $\neg(HVC_i < HVC_j) \wedge \neg(HVC_j < HVC_i)$, then the two hybrid vector clocks are concurrent, denoted as $HVC_i || HVC_j$.

If we set $\epsilon = \infty$, then hybrid vector clocks have the same properties as vector clocks. If ϵ is finite, certain entries in HVC_i can have the default value $PT_i - \epsilon$ and their representation can be compressed. For example, if $n = 10, \epsilon = 20$, a hybrid vector clock $HVC_0 = [100, 80, 80, 95, 80, 80, 100, 80, 80, 80]$ could be represented by $n(10)$ bits 10010010001 and a list of three integers $100, 95, 100$, instead of a list of ten integers.

Our monitors can work with either of these clocks. We use HVC in our implementation to facilitate its use when the number of processes is very large. However, in the experimental results, we ignore this optimization and treat as if ϵ is ∞.

2.1.5 A Basic Framework of Predicate Detection

The goal of a predicate detection algorithm is to ensure that the predicate of interest Φ is always satisfied during the execution of the distributed system. In other words, we want the monitors to notify us of cases where predicate Φ is violated ($\neg\Phi = true$). To detect whether the given predicate Φ is violated, we utilize the notion of *possibility* modality [49, 50] as described below.

Each process has a trace of its local events/states. From the traces of local states, Marzullo and Neiger [49] enumerate consistent global states by combining local states which are mutually concurrent. Then global states are connected together as follows: connect the global state S_1 to the global state S_2 if S_2 could be reached from S_1 by a single operation/event in some process (i.e. the cut corresponding to S_2 has exactly one more event than the cut corresponding to S_1). The graph of all consistent cuts and their connections forms a lattice as shown in Figure 2.1. The lattice shows all possible paths that the distributed system may execute.

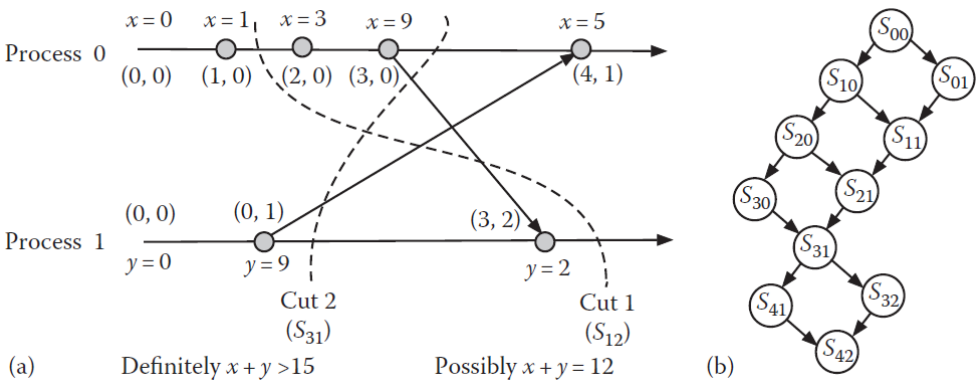


Figure 2.1: (This figure is based on [1]). The lattice of distributed computation history (right) is constructed from a particular execution (left) and causality. $x + y > 15$ is a *definitely* predicate since it is met by state S_{31} and every computation passes through S_{31}. $x + y = 12$ is a *possibly* predicate since it is met by only S_{21} and there are computation paths not going through S_{21}.

Denote Φ be the predicate under consideration. There are 3 possibilities [49]:

- *Possibly* Φ: there exists at least a global state in the lattice that satisfies Φ. In other words, there exists a valid distributed computation during which we can observe the truthification of Φ. If Φ represents the existence of a bug then possibly Φ implies the bug exists but we may or may not observe it during an execution of the distributed system.

- *Definitely* Φ: there exists a finite set of global states A such that: (1) all states in A satisfy Φ, and (2) all infinite distributed computations must pass through at least one state in A. If Φ represents the existence of a bug then definitely Φ implies the bug exists and we will certainly observe it during any long enough execution.

- *Never* Φ: no state in the lattice satisfies Φ. If Φ represents the existence of a bug then never Φ implies the distributed system is free from that bug.

The above framework can be used for global predicate detection of an arbitrary predicate. However, since it explores the state space thoroughly, this approach has worst-case exponential time complexity of $O(M^n)$ where n is the number of processes and M is the average number of events on each process. It can be proved that the problem of global predicate detection (GLOB) in general case is NP-hard [3]. First, GLOB is a decision problem because the goal is to determine

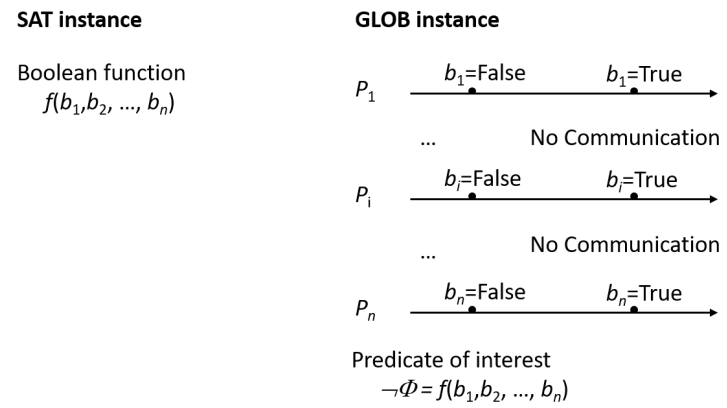


Figure 2.2: Reduction of a SAT instance to a GLOB instance.

whether there exists a consistent global state in which the predicate of interest Φ is violated ($\neg\Phi = true$. Given a consistent global state, we can evaluate Φ at that global state in polynomial time (with respect to the number of processes n). Therefore, GLOB is NP. The prove the problem is NP-complete, we reduce an arbitrary instance of SAT problem into an instance of GLOB. An instance of SAT problem is a Boolean function of $f(b_1, b_2, ..., b_n)$ where b_i are Boolean variables. The corresponding GLOB instance (cf. Figure 2.2 consists of n processes. On process P_i, there are only two internal events that assign $b_i = false$ and $b_i = true$ respectively. There is no communication between processes. The global predicate to be detected is $\neg\Phi = f(b_1, b_2, ..., b_n)$. It is clear that f is satisfiable if and only if there exists a consistent cut in which $\neg\Phi = true$.

Although the general global predicate detection problem is NP-complete, for some classes of predicates, there exist efficient detection algorithms. In this dissertation, we focus on two classes of predicates: linear predicates and semi-linear predicates. These predicates can be detected efficiently and they are commonly used in monitoring and verification of distributed systems. For example, conjunctive predicates are linear predicates while mutual exclusions are semi-linear predicates. In Sections 2.1.6 and 2.1.7, we recall the definitions of linear predicates, semilinear predicates and their detection algorithms [2, 3].

2.1.6 Linear Predicate and Detection Algorithm

In this section and the next sections, we recall the definitions of linear and semilinear predicates and their detection algorithms as described in [3]. Before we describe the linear predicates, we define some relevant definitions.

Path is a sequence of cuts where the later cut has more than previous cut exactly one local state. Path $S = X_1, ..., X_k$ such that $X_{i+1} = X_i \cup \{\text{one more state}\}$. For example, in Figure 2.3, the sequence C_1, C_2, C_4 is a path. The sequences C_1, C_4 and C_1, C_2, C_3, C_4 are not paths.

Suppose X and Y are two consistent global cut. We say Y is *reachable* from X, denoted as $X \leq Y$, if there is an execution that takes the system from X to Y. In other words, there exists a path in which X appears before Y. The notion of path and reachable is also applicable for consistent global states since each consistent cut corresponds to exactly one consistent global state.

Supremal state is a local state that is not less than any other local states of the cut. Intuitively, supremal states are the right-most states of the cut at each process. The set of all supremal states of a cut C is denoted as $sup(C)$. For example, in the example of Figure 2.3, we have $sup(C_1) = \{e_1, e_2\}$, $sup(C_2) = \{e_1, e_4\}$, $sup(C_3) = \{e_3, e_2\}$, and $sup(C_4) = \{e_3, e_4\}$.

Successor of a local state s is the next local state after s in the same process, denoted by $succ(s)$. If s is a local state in cut X, we define $X^s = X \cup succ(s)$. X^s is called the advance of cut X along local state s. For example, in Figure 2.3 $succ(e_1) = e_3$, $succ(e_2) = e_4$, $C_1^{e1} = C_3$, $C_1^{e2} = C_2$, $C_2^{e3} = C_4$, $C_3^{e4} = C_4$.

If the execution stops at cut/state X then X is called final, denoted as $final(X)$.

Possibly-Φ: We say that the predicate Φ is possibly satisfied between two state/cut X and Y, denoted as $P_\Phi(X, Y)$, if there exist an execution from X to Y along which Φ is satisfied at some state/cut.

$$P_\Phi(X, Y) \equiv \exists W : (X \leq W \leq Y) \wedge (\Phi(W) = true)$$

Definitely-Φ: We say that the predicate Φ is definitely satisfied between two state/cut X and Y,

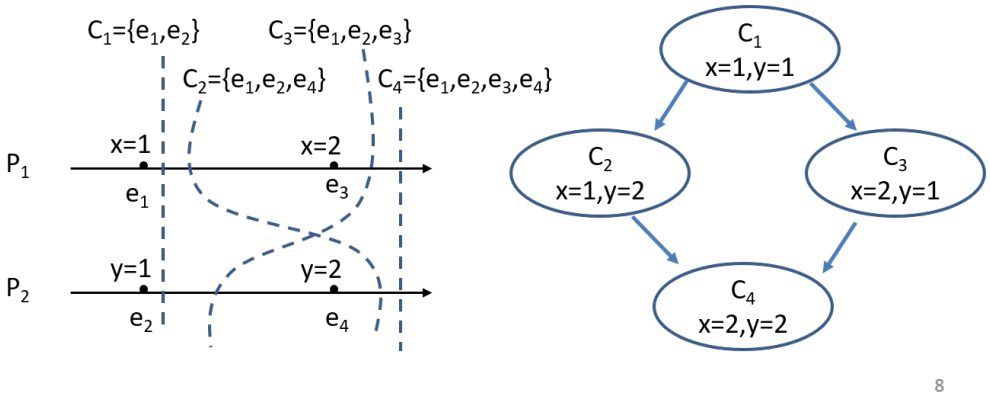


Figure 2.3: An example of a distributed computation

denoted as $D_\Phi(X, Y)$, if along any execution from X to Y, Φ is satisfied at some state/cut.

$$D_\Phi(X, Y) \equiv \forall S : (S \text{ is a path from } X \text{ to } Y) \wedge (\exists W \in S : \Phi(W) = true)$$

Table 2.1 provides some examples of possibly-Φ and definitely-Φ. It is obvious that $D_\Phi(X, Y) \Rightarrow P_\Phi(X, Y)$.

Table 2.1: Examples of Possibly-Φ and Definitely-Φ for the computation in Figure 2.3

Φ	$P_\Phi(C_1, C_4)$	$D_\Phi(C_1, C_4)$
$x + y = 3$	Yes	Yes
$y - x = 1$	Yes	No
$x + y = 4$	Yes	Yes
$x = 0$	No	No

Forbidden state: Suppose X is a consistent cut and $\Phi(X) = false$. A local state $s \in X$ is called a forbidden state, denoted as $forb_\Phi(s, X)$ when Φ will remain false until the cut leaves the forbidden state s.

$$forb_\Phi(s, X) \equiv \forall Y : X \leq Y : (\Phi(Y) = false) \vee (X^s \leq Y)$$

Linear predicate: predicate Φ is linear, denoted as $linear(\Phi)$, if for any cut X where $\Phi(X) = false$, at least one of the supremal states of X is a forbidden state.

$$linear(\Phi) \equiv \forall X : (\Phi(X) = false) \Rightarrow (\exists s \in sup(X) : forb_\Phi(s, X))$$

Thus to find a consistent global state satisfying Φ, we have to advance the cut passing by the forbidden state(s). However, we should not advance the cut passing by non-forbidden state(s) unless required. For the example in Figure 2.3 and let $\Phi = (x = 1) \wedge (y = 2)$. Φ is not satisfied in C_1. The forbidden state is e_2. If we advance C_1 passing e_2 we get C_2 in which Φ is satisfied. However, if we advance C_1 passing e_1 we get C_3. After than, even if we advance C_3 passing e_2 we will get C_4 and Φ is still not satisfied along that path. When we advance the cut along a local state, says e, the cut may become inconsistent because e is not concurrent with some existing supremal states. In this case, we have to advance all supremal states not concurrent with e. This process is repeated until all supremal states are concurrent. If we advance the cut a long a state e and after that the cut is still consistent, then e is called an eligible state of the cut. The set of all eligible states of a cut X is denoted as $eligible(X)$.

$$eligible(X) = \{s : s \in sup(X) \wedge X^s \text{ is consistent}\}$$

Algorithm 1 illustrates the algorithm for detection of linear predicate [2].

Algorithm 1 Linear predicate detection algorithm adapted from [2]

```
1:  Input:
2:      Φ                                   ▷ global linear predicate to detect
3:  Variable:
4:      GS                                  ▷ global state
5:  Initialization:
6:      GS ← set of initial local states
7:  while Φ(GS) == false do
8:      Find forbidden local state s ∈ GS
9:      GS ← GS ∪ succ(s)                   ▷ advance GS along s
10:     consistent(GS)                      ▷ make GS consistent
11: end while
12: return GS
```

We observe that cut GS is never moved backward. Each advance adds a new event to GS. Thus the total number of global states/cuts enumerated is at most the total number of events.

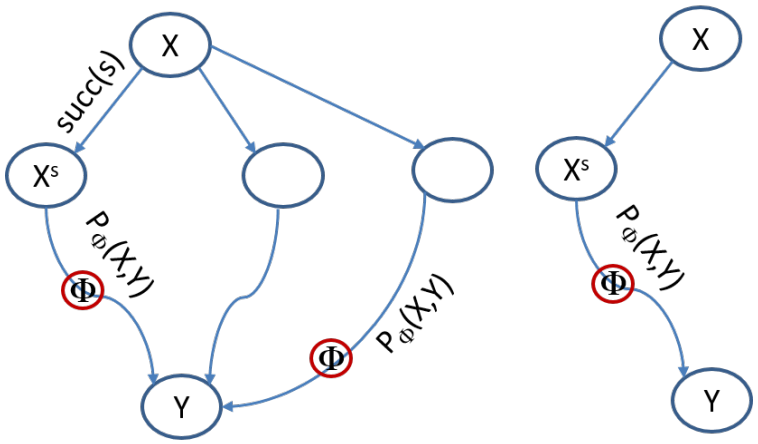


Figure 2.4: Illustration of semi-forbidden state.

2.1.7 Semilinear Predicate and Detection Algorithm

Semi-forbidden state: Suppose $\Phi(X) = false$. Local state $s \in X$ is called a semi-forbidden state, denoted as $sforb_\Phi(s, X)$, if

$$sforb_\Phi(s, X) \equiv \forall Y : X^s \leq Y : P_\Phi(X, Y) \Rightarrow P_\Phi(X^s, Y)$$

Figure 2.4 illustrates the meaning of a semi-forbidden state. If predicate Φ is ever satisfied along some path from X to Y, then it will be satisfied by some path from X^s to Y. Hence, by advancing the cut passing semi-forbidden state, we will find a global state where Φ is satisfied (if there is any such global state). We note that a semi-forbidden state is also an eligible state.

Semi-linear predicate: predicate Φ is semilinear, denoted as $semilinear(\Phi)$, if for any cut X where $\Phi(X) = false$, at least one of the supremal states of X is a semi-forbidden state state.

$$Semilinear(\Phi) \equiv \forall X : ((\Phi(X) = false) \wedge \neg final(X)) \Rightarrow \exists s \in eligible(X) : sforb_\Phi(s, X)$$

To find the global state where a semilinear predicate Φ is satisfied, we find the semi-forbidden state in the current cut and advance the cut along that state as illustrated in Algorithm 2.

We observe that cut X is never moved backward and each advance adds a new event to X. Thus the total number of global states/cuts enumerated is at most the total number of events. The procedure to find the semi-forbidden state depends on the nature of Φ. For example, if Φ is a

Algorithm 2 Semilinear predicate monitor algorithm adapted from [3]

Input:
  Φ ▷ global semilinear predicate to detect
Variable:
  GS ▷ global state
Initialization:
  $GS \leftarrow$ set of initial local states
while $\Phi(GS) == false$ **do**
  Find $s \in GS : s \in eligible(GS) \wedge sforb_{\Phi}(s, GS)$
  $GS \leftarrow GS \cup succ(s)$ ▷ advance GS along s
end while
return GS

mutual exclusion predicate, then finding the semi-forbidden state is equivalent to finding all eligible states [3]. Checking whether a local state s is eligible is $O(n)$ as we have to compare s with other supremal states. So the cost of finding the semi-forbidden state is $O(n^2)$.

2.2 Key-Value Store

2.2.1 General Architecture of a Key-Value Store

We utilize the standard architecture for key-value stores. Specifically, the data consists of (one or more) tables with two fields, a unique key and the corresponding value. The field value consists of a list of <*version, value*> pairs. A version is a vector clock that describes the origin of the associated value. It is possible that a key has multiple versions when different clients issue *PUT* (write) requests for that key independently. When a client issues a *GET* (read) request for a key, all existing versions of that key will be returned. The client could resolve multiple versions for the same key on its own or use the resolver function provided from the library. To provide efficient access to this table, it is divided into multiple partitions. Furthermore, to provide redundancy and ease of access, the table is replicated across multiple replicas.

To access the entries in this table, the client utilizes two operations: *GET* and *PUT*. The operation *GET* (x) provides the client with the value (or values if multiple versions exist) associated with key x. The operation *PUT* (x, val) changes the value associated with key x to val. The state

of the servers can be changed only by *PUT* requests from clients.

2.2.2 Voldemort Key-Value Store

Voldemort is LinkedIn's open source equivalence of Amazon's Dynamo key-value store. In Voldemort, clients are responsible for handling replication. When connecting to a server for the first time, a client receives meta-data from the server. The meta-data contains the list of servers and their addresses, the replication factor (N), required reads (R), required writes (W), and other configuration information.

When a client wants to perform a *PUT* (or *GET*) operation, it sends *PUT* (*GET*) requests to N servers and waits for the responses for a predefined amount of time (timeout). If at least W (R) acknowledgments (responses) are received before the timeout, the *PUT* (*GET*) operation is considered successful. If not, the client performs one more round of requests to other servers to get the necessary number of acknowledgments (responses). After the second round, if still less than W (R) replies are received, the *PUT* (*GET*) operation is considered unsuccessful.

Since the clients do the task of replication, the values N, R, W specified in the meta-data is only a suggestion. The clients can change those values for their needs. By adjusting the value of W, R, and N, the client can tune the consistency model. For example, if $W + R > N$ and $W > \frac{N}{2}$ for every client, then they run on sequential consistency. On the other hand, if $W + R \leq N$ then they have eventual consistency. For example, when $N = 3, R = 1, W = 1$ we have eventual consistency. When $N = 3, R = 1, W = 3$ or $N = 3, R = 2, W = 2$, we have sequential consistency. In this dissertation, we also use $NxRyWz$ (or $RyWz$ when value of N is clear from the context) as the short form of $N = x, R = y, W = z$.

2.2.3 The Performance Difference between Eventual and Sequential Consistency in Voldemort Key-Value Store

Let us consider the *PUT* request first. In reality, a *PUT* request consists of a *GET_VERSION* request (to obtain the latest version of the key so that a new version value can be issued properly)

and an actual *PUT* request (to request the replicas to commit to their database the new value and new version of the key). However, we assume a *PUT* request is an actual *PUT* request for now and we will discuss the full *PUT* request later. When a client issues a *PUT* request, it needs to receive enough number of acknowledgments from replicas/servers within the timeout to succeed. In sequential consistency, the required number of "in-time" acknowledgments is higher. Thus, the client has to wait for acknowledgments from more servers, especially the ones in the remote distance. Due to network latency, it is possible that the acknowledgments from the remote servers arrive late after the timeout and the client has to re-issue the *PUT* requests (the late acknowledgments are ignored by the client). This increases the latency and decreases the throughput of *PUT* request. On the other hand, in eventual consistency, the number of "in-time" acknowledgment is lower (typically one) and the client request is likely to be satisfied promptly by the local server. Thus, the latency of a *PUT* request in eventual consistency is low and the throughput is high.

We note that in eventual consistency, even if the acknowledgments from remote servers arrive late, the *PUT* request is still committed at remote databases as long as the servers receive the *PUT* request. Data anomalies (the *PUT* request is committed at some servers and not at some other servers) occur if the *PUT* request fails to arrive some servers because of network or hardware failures. This scenario is expected to be rare. So intuitively, since the eventually consistent client knows that its *PUT* request is very likely to be committed at every server, the client believes it is safe to proceed without waiting for confirmations from remote servers after the timeout. On the other hand, the sequentially consistent client thinks that it is only safe to proceed if the confirmations are received in time. Otherwise, the client has to re-issue the request to make sure the servers are consistent. This cautiousness increases the latency (and bandwidth) and decreases the throughput in exchange for the guarantee of consistency.

For *GET* requests, it is similar. An eventually consistent client thinks the value stored at the local server is identical to that stored at remote servers, thus it proceeds without waiting for responses from remote servers after the timeout. The problem occurs if different servers store different versions for the same key (because of anomalies caused by *PUT* requests). In this case,

different clients can read different values of the same key, which may result in incorrect actions by the clients.

Regarding the full *PUT* request, it consists of two independent requests: a *GET_VERSION* request (which is basically similar to a *GET* request) and an actual *PUT* request. Thus compared to a *GET* request, the chance for a full *PUT* request to succeed (i.e. the client receives replies for both *GET_VERSION* and actual *PUT* from servers within the timeout) is lower. Consequently, the performance difference between eventual and sequential consistency is larger for (full) *PUT* requests than for *GET* requests.

We also observe another scenario for data anomaly with a full *PUT* request as follows. Suppose we have two servers S_1 and S_2 at locations distant from each other. Client C_1 and client C_2 are close to S_1 and S_2 respectively. Assume the key-value store is eventually consistent. Consider the key x with original version $< ver = 0, val = 0 >$. Client C_1 wants to update x with value 1. It issues *GET_VERSION* requests, receives the reply from S_1, and ignores the expired reply from S_2. Then it issues the actual *PUT* request with new value $< ver = 1, val = 1 >$ for key x. Client C_2 simultaneously does the same but with new value$< ver = 1, val = 2 >$. When server S_1 receives the actual *PUT* request from C_1, it commits the request to its database. When the actual *PUT* request from C_2 arrives S_1 later, the request is rejected because server S_1 observes the version ($ver = 1$) is not newer than the current version of x in its database. In a symmetrical way, the value $< ver = 1, val = 2 >$ is committed to S_2 database. Assume the reject responses from S_1 to C_2 and from S_2 to C_1 arrive late and are ignored by the clients. Then the clients will proceed while leaving the server databases inconsistent. If sequential consistency is used, the clients will observe the reject responses from the servers and they will re-issue the requests until positive acknowledgments are confirmed. Although that guarantees a consistent database, the repeated requests also increase the latency observed by the application at the higher level.

Late arrivals of replies from remote servers is probably the main reason for the decreased performance in sequential consistency because they cause the clients to retry. Consequently, when network latency increases (for example the replicas/servers are distributed over a wider area for

higher fault-tolerance and availability), the performance difference between eventual and sequential consistency is expected to be larger. We also note that different sequential consistency configurations have different performance. For example, $R1W3$ is better than $R2W2$ for *GET* requests because $R2W2$ has stricter requirement for *GET* requests. However, $R1W3$ is worse than $R2W2$ for *PUT* requests. As a result, in an application where *PUT* requests dominate, $R1W3$ is better than $R2W2$. Since it is typical that *PUT* requests constitute the majority in an application, we usually choose configurations like $R = 1, W = N$ as representatives for sequential consistency.

In summary, we anticipate that the following factors can influence the performance difference between eventual and sequential consistency:

- Workload characteristics. The difference is larger for *PUT* request than for *GET* request. Thus when the percentage of *PUT* requests increases, the difference is increased.
- Network latency. When network latency is larger, the difference increases.
- Specific sequential consistency configuration. For example, $R1W3$ is expected to perform better than $R2W2$ in a typical application.

We will validate our hypothesis in Chapter 3.

2.3 Distributed Programs

A program p consists of a set of nodes V_p and a set of edges E_p. We assume that for any node j, edge (j, j) is included in E_p. Each node, say j, in V_p is associated with a set of variables var_j. The set of variables of program p, denoted by var_p, is obtained by the union of the variables of nodes in p. A state of p is obtained by assigning each variable in var_p a value from its domain. State space of p, denoted by S_p, is the set of all possible states of p.

Each node j in program p is also associated with a set of actions, say ac_j. An action in ac_j is of the form $g \longrightarrow st$, where g is a predicate involving $\{var_k : (j, k) \in E_p\}$ and st updates one or more variables in var_j. We say that an action ac (of the form $g \longrightarrow st$) is enabled in state s if and only if g evaluates to true in state s. The transitions of action ac (of the form $g \longrightarrow st$)

are given by $\{(s_0, s_1) |\ s_0, s_1 \in S_p$, g is true in s_0 and s_1 is obtained by execution st in state $s_0\}$. Finally, transitions of node j (respectively, program p) is the union of the transitions of its actions (respectively, its nodes). We use δ_{ac}, δ_j and δ_p to denote transitions corresponding to action ac, node j and program p respectively.

2.3.1 Traditional/Active-Node Model

Computation. In the traditional/active-node model, the computation of program p is of the form $\langle s_0, s_1, \cdots \rangle$ where

- $\forall l : l \geq 0 :$, s_l is a state of p,
- $\forall l : l \geq 0 : (s_l, s_{l+1})$ is a transition of p or
  $((s_l = s_{l+1})$ and no action of p is enabled in state $s_l)$, and
- If some action ac of p (of the form $g \longrightarrow st$) is continuously enabled (i.e., there exists l such that g is true in every state in the sequence after s_l) then ac is eventually executed (i.e., for some $x \geq l$, (s_x, s_{x+1}) corresponds to execution of st.)

The above computation model corresponds to the centralized daemon with interleaving semantics wherein each step, only one node can execute at a given time. This can be implemented in read-write atomicity or message passing model by solutions such as local mutual exclusion, dining philosophers, etc. The resulting computation guarantees that two neighboring nodes do not execute simultaneously. In turn, the resulting computation is realizable in the original model. (Our observations/results are also applicable to other models. We discuss this in Section 4.4.)

2.3.2 Passive-Node Model

The structure of the program (in terms of its nodes and actions) remains the same in the passive-node model. The only difference is in terms of the execution model. Specifically, the system consists of a replicated and partitioned key-value store that captures the current state of p. In other words,

the state of p is stored in terms of pairs of the form $\langle k, v \rangle$, where k is a key (i.e., the name of the variable and node ID) and v is the corresponding value. Additionally, the system contains a set of clients. The role of the clients is to execute the actions of one or more nodes assigned to them (either statically or dynamically).

In an ideal environment, the execution of the program is performed as follows: Let node j be assigned to client $c1$. Then, $c1$ reads the values of the variables of j and its neighbors. If it finds that some action of j is enabled, it updates the key-value store with the new values for the variables of j. Similar to active-node model, it is required that the actions of multiple nodes can be serialized.

Computation. The notion of computation in the passive-node model is identical to that of the active-node model from Section 2.3.1; the only difference is the introduction of clients in the passive-node model. Furthermore, by requiring the clients to execute actions of each node infinitely often, it guarantees the fairness assumed in the definition of computation in Section 2.3.1.

2.3.3 Similarity between Active-Node and Passive-Node Model

The passive-node model relies on two requirements (1) each node is given a fair chance to execute, and (2) execution corresponds to a sequence of atomic executions of actions of some nodes. The first requirement is satisfied as long as each client considers every node infinitely often; if some action is enabled continuously, eventually a client would execute that action. The second requirement, atomicity of individual actions, is satisfied if (1) clients enforce local mutual exclusion among nodes, i.e., if we ensure that clients $c1$ and $c2$ do not operate simultaneously on nodes j and k that are neighbors of each other and (2) when a client reads the value of any variable (key), it obtains the most recent version of that variable (key).

Of these, the requirement for mutual exclusion was necessary even in the active-node model. The ability to read the most recent value was inevitable in the active-node model. Specifically, if node j reads the values of its neighbors after it had acquired the local mutual exclusion, it was

guaranteed to read the latest state of its neighbors. In the passive-node model, this requirement would be satisfied if we have only one data store (i.e., no replication) that maintains the data associated with all nodes or the replicated data store *appears* as a single copy. In particular, if the replicated data store provides a strong consistency such as sequential consistency, this property is satisfied. However, if it provides a weaker consistency such as eventual consistency, this property may be violated. We discuss the details of this sequential/eventual consistency, next in Section 2.4.

2.3.4 Executing a Node Action by Client

The procedure for a client C to process a node i assigned to its partition is as follows:

(1) Obtain exclusive update privilege for the state of node i and read privilege for neighbors of i (i.e. no other client should read the state of i or update the state of i neighbors).

(2) Read the state of i (variables of i) and its neighbors.

(3) Compute the new values for i's variables.

(4) Write the new state of i to the store (this step can be omitted if all of i's variables are unchanged), and

(5) Release the privileges it holds for i and its neighbors.

In order to support client C in obtaining necessary privileges, we designate one Peterson lock [13] associated with each graph edge. To read the state of node i, client C just needs to obtain a lock associated with any edge incident on i. To update the state of node i, however, client C needs to obtain the locks associated with all edges incident on i. Once such locks are obtained by C, other clients can read state of i's neighbors but they cannot update any of them (since they have to wait for one of the locks being hold by C).

For deadlock avoidance, client C obtains the required locks in lexicographical order. Suppose $i < j$ then the lock for edge (i, j) is L_i_j. As an illustration, in Figure 2.5, if client C wants to update node 6, it has to obtain these locks in the following order L_1_6, L_5_6, L_6_9.

Once C has had the update privilege for node 6, no other client can update any neighbor of 6 (says 5) since that client will have to wait for C to release the lock L_5_6.

We note that the above locking scheme only works if the shared data is sequentially consistent. If it is eventually consistent, simultaneous updates could happen as explained in Section 2.4.

Since obtaining locks constitutes a sizeable proportion of client computation time (waiting for other clients to release the required locks) and many nodes shared neighbors, a client usually does not process each node individually but processes a batch of nodes at the same time. This batch processing reduces the number of locks the client needs to obtain. For example, suppose nodes 5 and 6 are assigned to a client. If the client processes 5 and 6 individually, it will have to obtain 5 locks, namely L_1_5, L_5_9, L_5_20 (for node 5) and L_1_6, L_6_9 (for node 6). Note that there is no need for obtaining lock L_5_6 if both nodes are assigned to the same client. On the other hand, if the client processes node 5 and 6 together, it only has to obtain 3 locks, namely either L_1_6 or L_1_6, L_5_20, and either L_5_9 or L_6_9.

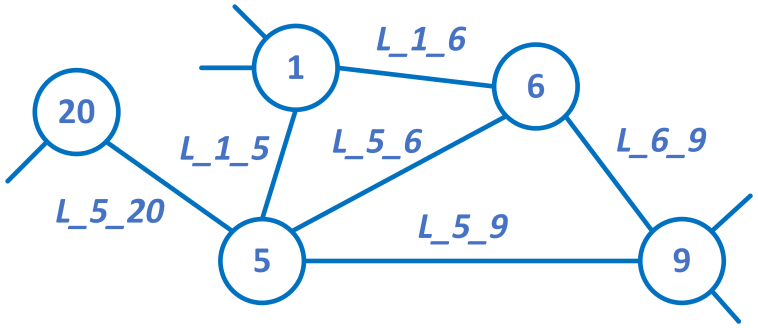


Figure 2.5: Illustration of locks. To update node 6, a client has to obtain these locks in following order: L_1_6, L_5_6, L_6_9

2.3.5 Stabilization

In this section, we recall the definition of stabilization from [53]. This definition relies on the notion of computation. As discussed in Section 2.3.1 and 2.3.2, computations can be defined in both active-node model and passive-node model. We use this notion of computation in defining stabilization.

Stabilization. Let p be a program. Let I be a subset of state space of p. We say that p is stabilizing with state predicate I if and only if

- *Closure*: If program p executes a transition in a state in I then the resulting state is in I, i.e., for any transition $(s_0, s_1) \in \delta_p$, $s_0 \in I \Rightarrow s_1 \in I$, and
- *Convergence*: Any computation of p eventually reaches a state in I, i.e., for any $\langle s_0, s_1, \cdots \rangle$ that is a computation of p, there exists l such that $s_l \in I$.

In our context, we use I to capture the predicate to which program recovers so that the subsequent computation satisfies the specification. We use the term *invariant* of p to denote this predicate. In our *initial* discussion, we focus only on the convergence property. Hence, we focus on *silent stabilization* which requires that upon reaching the invariant, the program terminates, i.e., it has no further actions that it can execute. Thus, we have

Silent Stabilization. Let p be a program. Let I be a subset of state space of p. We say that p is silent stabilizing with state predicate I if and only if

- *Closure*: Program p has no transitions that can execute in I, i.e., for any $s_0 \in I$, $(s_0, s_1) \notin \delta_p$ for any state s_1, and
- *Convergence*: Any computation of p eventually reaches a state in I, i.e., for any $\langle s_0, s_1, \cdots \rangle$ that is a computation of p, there exists l such that $s_l \in I$.

Our initial discussion focuses on silent stabilization. We discuss generalized stabilization in Section 4.4.

2.4 Consistency Violating Faults (cvf)

In the passive-node model, the program state is stored at the replicas. The protocol for synchronizing replicas can be passive replication or active replication (the case of Voldemort).

In passive-replication-based sequential consistency, the protocol enforces that all replicas are strictly synchronized. A replica will not provide the new value unless that value has been committed by other replicas. A client reading from any of the replicas will always obtain the fresh data. However, for eventual consistency, the protocol is relaxed and allows replicas to return the current values they know, which may be not up-to-date.

In active-replication-based sequential consistency, the protocol requires each update to be committed by a majority of replicas and the client reads from at least one of them, thus obtains the fresh data. For eventual consistency, however, the protocol is relaxed where the read and write quorums do not overlap. Thus, some replicas may have not received the latest updates due to transient faults, and if a client reads from those replicas, it obtains a stale value.

In short, a client always obtains the fresh data with sequential consistency and may obtain a stale data with eventual consistency. Reading stale information could lead the clients to incorrect computation steps/transitions.

Abstract description of cvf. For each variable x of node j, each replica i maintains a value of $x.j$ as a key-value pair. For the purpose of illustration, assume that there are three replicas and the values of $x.j$ at these replicas are r_1, r_2 and r_3. Denote $f(r_1, r_2, r_3)$ as the abstract value of $x.j$ where f is some resolution function that chooses a value among r_1, r_2, r_3 in a deterministic manner. For example, function f chooses the latest value of $x.j$ (assume that each value is also associated with a logical or physical timestamp). In sequential consistency where the replication protocol provides the impression that all the replicas work as if there is only a single replica, access (read/write) to variable $x.j$ by any client always returns the same abstract value of $x.j$. In eventual consistency, however, this property may be violated when different clients observe different values of $x.j$ (e.g. client $c1$ observes value r_1 while client $c2$ observes value r_2). Only one of those values is up-to-date and the other is stale. We also note that reading stale data is possible in eventual consistency but such anomalies are expected to be not frequent [4] and they are usually associated with transient faults.

Reading stale values due to eventual consistency in Voldemort. Figure 2.6 illustrates how a cvf occurs in Voldemort key-value store where the clients are running on eventual consistency R1W1. Suppose $x = 0$ initially. Client 1 updates the value of x to 1 by sending PUT(x, 1) request to all replicas/servers. Due to a temporary network failure, the request does not reach server 3. However, the PUT request still succeeds since client 1 receives replies from two servers (PUT request is successful if client receives at least W=1 replies). If client 3 reads the value of x, it will obtain the stale value $x = 0$ from server 3 (this read succeeds since client 3 just needs R=1 response). In contrast, any client served by server 1 and server 2 will see the new value $x = 1$.

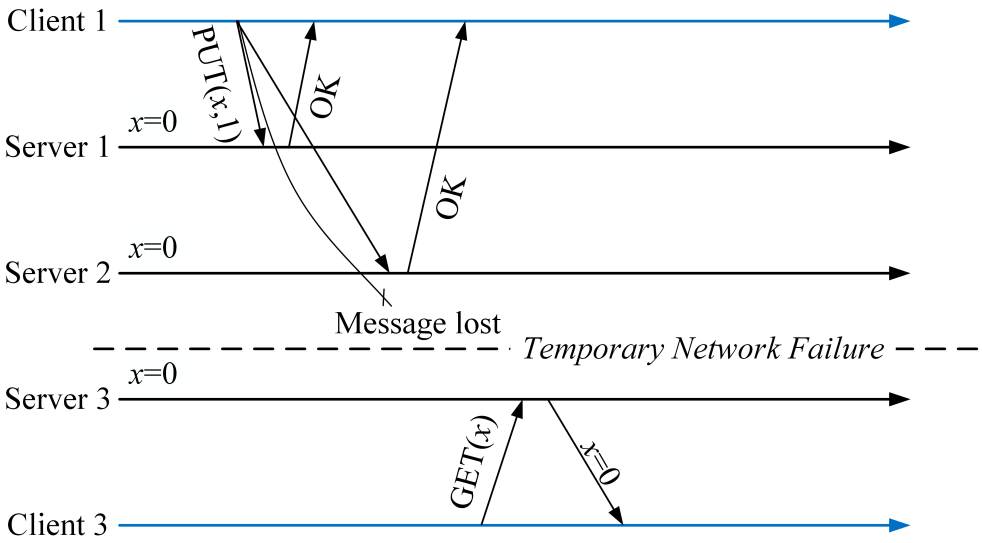


Figure 2.6: Illustration of cvf in Voldemort. Clients run on eventual consistency R1W1

Reading stale values can lead to erroneous transitions. For example, in the graph coloring problem suppose client 1 wants to work on node 5 while client 3 wants to work on node 20 (cf. Figure 2.5). If variable x in Figure 2.6 is the shared lock L_5_20, then client 1 will think that it has obtained the lock while client 3 observes the lock is still vacant and tries to obtain it (it will succeed since one confirmation from server 3 is sufficient). As a result, both clients enter a critical section simultaneously. Suppose the initial color of each node is color 0 and the two clients read these values. Then the clients will likely update the colors of both node 20 and node 5 to color 1, resulting in a new invalid coloring. We note that if locks are not used to guarantee atomicity (such

as in aggressive stabilization), client 1 and client 3 may also read and update the color of node 5 and node 20 simultaneously without knowing so, and produce invalid coloring results in a similar manner.

Consistency Violating Faults (cvf). With this observation, we can now view the computation of the given program p as a sequence $\langle s_0, s_1, \cdots \rangle$ such that most transitions $(s_l, s_{l+1}), l \geq 0$ in this sequence correspond to the transitions of p. However, some transitions correspond to the scenario where some node j reads an inconsistent (stale) value for some variable and updates one or more variables of j. By design, the incorrect execution corresponds to changing one or more variables of one node. Thus, the effect of the incorrect execution is state perturbation of some node. We denote such incorrect execution as concurrency violating faults (cvf_p)):

$cvf_p \subset \{(s_0, s_1) | s_0, s_1 \in S_p$ and s_0, s_1 differ only in the variables of some node j of $p\}$.

Remark 1. Whenever p is clear from the context, we use cvf instead of cvf_p.

Remark 2. If the clients do not utilize a mechanism to guarantee atomicity, they may read unreliable values that are not supposed to be read (values being updated by other clients), and incorrectly calculate values for some variables of some node, e.g. node j. When these incorrect values are updated to the store, that update has the same effect as perturbing the variables of node j. In other words, the incorrect transitions caused by violations of the action atomicity requirement can also be treated as cvfs.

Computation in the presence of cvf. With the definition of cvf, we can see that computation of program p in a given replicated passive-node model is of the form $\langle s_0, s_1, \cdots \rangle$ where

- $\forall l : l \geq 0 :$, s_l is a state of p,
- $\forall l : l \geq 0 : (s_l, s_{l+1}) \in \delta_p \cup cvf_p$ or

$(s_l = s_{l+1})$ and no action of p is enabled in state s_l, and

- If some action ac of p (of the form $g \longrightarrow st$) is continuously enabled (i.e., there exists l such that g is true in every state in the sequence after s_l) then ac is eventually executed (i.e., for some $x \geq l$, (s_x, s_{x+1}) corresponds to execution of st.)

Figure 2.7 illustrates a computation with cvfs.

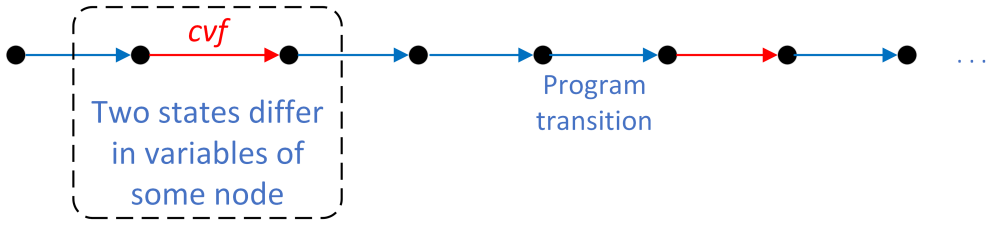


Figure 2.7: A computation in the presence of cvf

CHAPTER 3

DETECT-ROLLBACK APPROACH

In this chapter, we evaluate the benefits of the detect-rollback approach in handling cvfs. The correctness of this approach relies on two components: detection and rollback. In the detection phase, we run applications on eventual consistency and use the monitors to detect any violations that occur. Once a violation is reported, a rollback mechanism is invoked to recover the computation to a previous correct state and then resume the computation.

This chapter is organized as follows. Section 3.1 presents our design and implementation of the predicate detection module. Section 3.2 discusses some rollback mechanisms, presents our application-specific rollback algorithm as well as strategies for handling livelocks. Section 3.3 describes the experiment setups, the test cases we used, and analysis of experiment results. In particular, we are interested in evaluating the overhead and effectiveness of the monitors, the benefits of eventual consistency with monitors vs. sequential consistency, the benefit of detect-rollback (i.e. eventual consistency with monitors and rollback) vs. sequential consistency. We also interested in experiment results that help checking our anticipation in Section 2.2.3. We make concluding remarks in Section 3.4.

3.1 Predicate Detection Module

3.1.1 Overall Architecture

The predicate detection module (monitoring module) is responsible for monitoring and detecting violation of the global predicate of interest in a distributed system. The structure of the module is as shown in Figure 3.1. It consists of local predicate detectors attached to each server and the monitors independent of the servers. The local predicate detector caches the state of its host server and sends information to the monitors. This is achieved by intercepting the PUT requests for variables that may affect the predicate being monitored. The monitors run predicate detection algorithm based

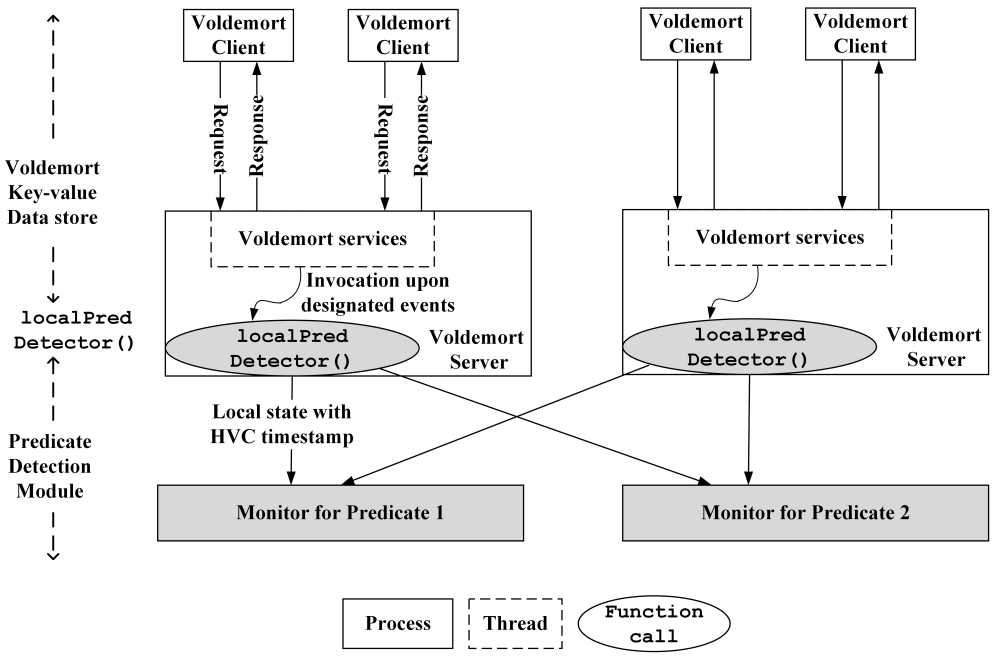


Figure 3.1: Architecture of predicate detection module

on the information received to determine if the global predicate of interest $\mathcal{P}$ has been violated.

We anticipate that the predicate of interest $\mathcal{P}$ is a conjunction of all constraints that should be satisfied during the execution. In other words, $\mathcal{P}$ is of the form $\mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \cdots \mathcal{P}_l$ where each $\mathcal{P}_i$ is a constraint (involving one or more processes) that the program is expected to satisfy. Each $\mathcal{P}_i$ can be of different types (such as linear or semilinear). The job of the monitoring module is to identify an instance where $\mathcal{P}$ is violated, i.e., to determine if there is a consistent cut where $\neg\mathcal{P}_1 \vee \neg\mathcal{P}_2 \vee \cdots \neg\mathcal{P}_l$ is true. In order to monitor multiple predicates, the designer can have multiple monitors with one monitor for each predicate $\mathcal{P}_i$ or one monitor for all predicates $\mathcal{P}_i$'s. In the former case, the detection latency is small but the overheads can be unaffordable when the number of predicates is large since we need many monitor processes. In the latter case, the overhead is small but the detection latency is long. We adopt a compromise: our monitoring module consists of multiple monitors and each monitor is responsible for multiple predicates. The predicates are assigned to the monitors based on the hash of the predicate names in order to balance the monitors' workload.

The number of monitors equals the number of servers and the monitors are distributed among the machines running the servers. We have done so to ensure that the cost of the monitors is

```
<predicate>
  <type>semilinear</type>
  <conjClause>
    <id>0</id>
    <var>
      <name>x2</name> <value>1</value>
    </var>
    <var>
      <name>y2</name> <value>1</value>
    </var>
  </conjClause>
  <conjClause>
    <id>1</id>
    <var>
      <name>z2</name> <value>1</value>
    </var>
  </conjClause>
</predicate>
```

Figure 3.2: XML specification for $\neg\mathcal{P} \equiv (x_1 = 1 \wedge y_1 = 1) \vee z_2 = 1$

accounted for in experimental results while avoiding overloading a single machine. An alternative approach is to have monitors on a different machine. In this case, the trade-off is between CPU cycles used by the monitors (when monitors are co-located with servers) and communication cost (when monitors are on a different machine). Our experiments suggest that in the latter approach (monitors on a different machine) monitoring is more efficient. However, since there is no effective way to compute the increased cost (of machines in terms of money), we report results where monitors are on the same machines as the servers.

Each (smaller) predicate $\mathcal{P}_i$ is a boolean formula on the states of some variables. Since any boolean formula can be converted to a disjunctive normal form, users can provide the predicates being detected ($\neg\mathcal{P}_i$'s) in disjunctive normal form. We use the XML format to represent the predicate. For example, the semilinear predicate, says $\neg\mathcal{P}_1 \equiv (x_1 = 1 \wedge y_1 = 1) \vee z_2 = 1$, in XML format is shown in Figure 3.2. Observe that this XML format also identifies the type of the predicate (linear, semi-linear, etc.) so that the monitor can decide the algorithm to be used for detection.

3.1.2 Local Predicate Detector

Upon the execution of a PUT request, the server calls the interface function `localPredicateDetector` which examines the state change and sends a message (also known as a candidate) to one or more monitors if appropriate. Note that not all state changes cause the `localPredicateDetector` to send candidates to the monitors. The most common example of this is when the changed variable is not relevant to the predicates being detected. Other examples depend upon the type of predicate being detected. As an illustration, if predicate $\neg\mathcal{P}$ is of the form $x_1 \wedge x_2$ then we only need to worry about the case where x_i changes from $false$ to $true$.

A candidate sent to the monitor of predicate $\mathcal{P}_i$ consists of an HVC interval and a partial copy of the server local state containing variables relevant to $\mathcal{P}_i$. The HVC interval is the time interval on the server when $\mathcal{P}_i$ is violated, and the local state has the values of variables which make $\neg\mathcal{P}_i$ true.

For example, assume the global predicate of interest to be detected is $\neg\mathcal{P} \equiv \neg\mathcal{P}_1 \vee \neg\mathcal{P}_2 \cdots \vee \neg\mathcal{P}_m$ where each $\neg\mathcal{P}_j$ is a smaller global predicate. Assume that monitor M_j is responsible for detection of predicate $\neg\mathcal{P}_j$. Consider a smaller predicate, says $\neg\mathcal{P}_2$, and for the sake of the example, assume that it is a conjunctive predicate, i.e. $\neg\mathcal{P}_2 \equiv (\neg LP_2^1) \wedge (\neg LP_2^2) \wedge ...(\neg LP_2^n)$ where n is the number of servers. We want to detect when $\neg\mathcal{P}_2$ becomes true. On a server, say server i, the local predicate detector will monitor the corresponding local predicate $\neg LP_2^i$ (or $\neg LP_2$ for short, in the context of server i as shown in Figure 3.3). Since $\neg\mathcal{P}_2$ is true only when all constituent local predicates are true, server i only has to send candidates for the time interval when $\neg LP_2$ is true. In Figure 3.3, upon the first PUT request, no candidate is sent to monitor M_2 because $\neg LP_2$ is false during interval $[HVC_i^0, HVC_i^1]$. After serving the first PUT request, the new local state makes $\neg LP_2$ true, starting from the time HVC_i^2. Therefore upon the second PUT request, a candidate is sent to monitor M_2 because $\neg LP_2$ is true during the interval $[HVC_i^2, HVC_i^3]$. This candidate transmission is independent of whether $\neg LP_2$ is true or not after the second PUT request is served. It depends on whether $\neg LP_2$ is true after the execution of the previous PUT request. That is why, upon the second PUT request, a candidate is also sent to monitor M_3 but none is sent to M_1. However, if

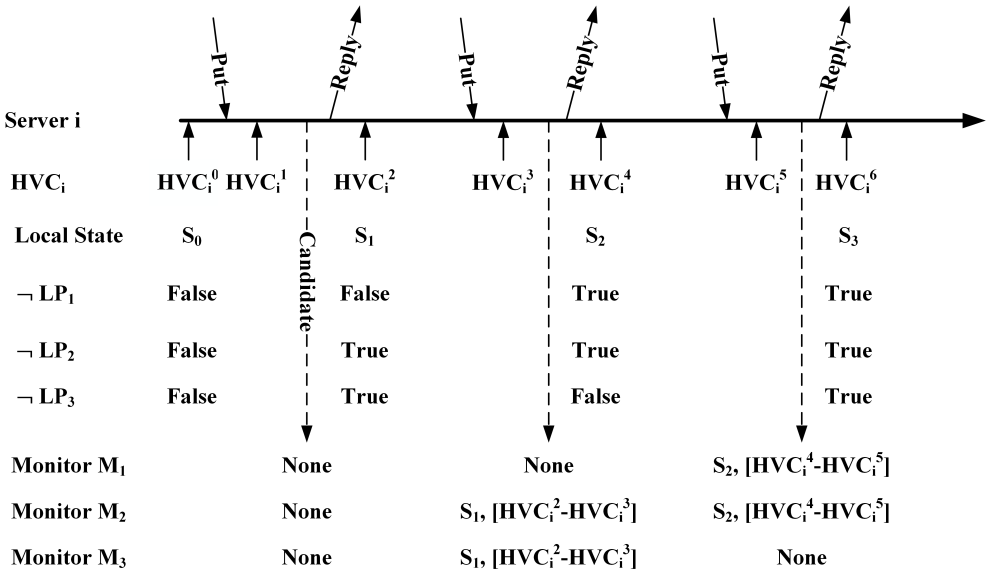


Figure 3.3: Illustration of candidates sent from a server to monitors corresponding to three conjunctive predicates. If the predicate is semilinear, the candidate is always sent upon a PUT request of relevant variables.

the predicate is not a linear predicate, then upon a PUT request for a relevant variable, the local predicate detector has to send a candidate to the associated monitor anyway.

3.1.3 Implementation of the Monitors.

The task of a monitor is to determine if some smaller predicate $\mathcal{P}_i$ under its responsibility is violated, i.e., to detect if a consistent state on which $\neg\mathcal{P}_i$ is true exists in the system execution. The monitor constructs a global view of the variables relevant to $\mathcal{P}_i$ from the candidates it receives. The global view is valid if all candidates in the global view are pairwise concurrent.

The concurrence/causality relationship between a pair of candidates is determined as follows: suppose we have two candidates $Cand_1, Cand_2$ from two servers S_1, S_2 and their corresponding HVC intervals $[HVC_1^{start}, HVC_1^{end}]$, $[HVC_2^{start}, HVC_2^{end}]$. Without loss of generality, assume that $\neg(HVC_1^{start} > HVC_2^{start})$ (cf. Figure 3.4).

- If $HVC_2^{start} < HVC_1^{end}$ then the two intervals have common time segment and $Cand_1 \| Cand_2$.
- If $HVC_1^{end} < HVC_2^{start}$, and $HVC_1^{end}[S_1] \leq HVC_2^{start}[S_2] - \epsilon$ then interval one is consid-

ered happens before interval two. Note that $HVC[i]$ is the element corresponding to process i in HVC. In this case $Cand_1 \rightarrow Cand_2$

- If $HVC_1^{end} < HVC_2^{start}$, and $HVC_1^{end}[S_1] > HVC_2^{start}[S_2] - \epsilon$, this is the uncertain case where the intervals may or may not have common segment. In order to avoid missing possible violations, the candidates are considered concurrent.

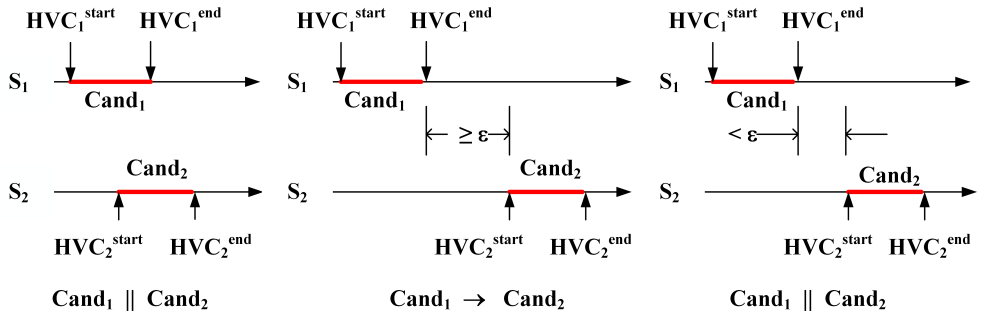


Figure 3.4: Illustration of causality relation under HVC interval perspective

When a global predicate is detected, the monitor informs the administrator or triggers a designated process of recovery. We develop detection algorithms for the monitors of linear predicates and semilinear predicates based on [2, 3] as shown in Algorithm 1 and Algorithm 2 (these Algorithms are described in Section 2.1). Basically, the algorithms have to identify the correct candidates to update the global state (GS) so that we would not have to consider all possible combinations of GS as well as not miss the possible violations. In linear (or semilinear) predicates, these candidates are forbidden (or semi-forbidden) states. Forbidden states are states such that if we do not replace them, we would not be able to find the violation. Therefore, we must advance the global state along forbidden states. Semi-forbidden states are states such that if we advance the global state along them, we would find a violation if there exists any. The procedure of advancing the global snapshot GS along a local state s (s belongs to GS) means the successor of s is added to GS. The successor of a local state s is the next local state after s on the same process. As s is replaced by its successor, the global snapshot GS "advances" forward. When advancing global state along a candidate, that candidate may not be concurrent with other candidates existing in the global state. In that case,

we have to advance the candidates to make them consistent. This is done by `consistent(GS)` in the algorithm. If we can advance global state along a candidate without calling `consistent(GS)`, that candidate is called an eligible state. The set of all eligible states in the global state is denoted as `eligible(GS)` in the algorithms. For a more detailed discussion of linear and semi-linear predicates, we refer to [3].

After a consistent global state GS is obtained, we evaluate whether predicate $\mathcal{P}$ is violated at this global state ($\mathcal{P}(GS) = true$ means $\mathcal{P}$ is satisfied, $\mathcal{P}(GS) = false$ means $\mathcal{P}$ is violated). If $\mathcal{P}$ is violated, the algorithms return the global snapshot GS as the evidence of the violation. Note that the monitors will keep running even after a violation is reported so that possible violations in the future will not be missed. This is the case when the applications, after being informed about the violation and rolling back to a consistent checkpoint before the moment when the violation occurred, continue their execution and violations occur again. Hence the monitors have to keep running in order to detect any violations of $\mathcal{P}$.

The way we evaluate $\mathcal{P}$ on global state GS is slightly different from the algorithms in [2,3,54,55]. In those algorithms, the candidates are sent directly from the clients containing the states of the clients. In our algorithms, the candidates are sent from the servers containing the information the servers know about the states of the clients that have been committed to the store by the clients. Note that, in a key-value store, the clients use the server store for sharing variables and committing updates. Therefore, the states of clients will eventually be reflected at the server store. Since the predicate $\mathcal{P}$ is defined over the states of the clients, in order to detect violations of $\mathcal{P}$ from the states stored at the server, we have to adapt the algorithms in [2, 3, 54, 55] to consider that difference. Furthermore, the state of a client can be stored slightly differently at different servers. For example, a PUT request may be successful at the regional server but not successful at remote servers. In that case, assuming we are using eventual consistency, the regional server store will have the update while remote stores do not have the update. Our algorithms also consider this factor when evaluating $\mathcal{P}$. For example, suppose variable x has version v_1 at a server and version v_2 at another server. Suppose that if $x = v_1$ then $\mathcal{P}$ is violated, and if $x = v_2$ then $\mathcal{P}$ is satisfied. To

avoid missing possible violations, our algorithms check all available versions of x when evaluating $\mathcal{P}$.

Since our algorithms are adapted from [2, 3, 54, 55], the correctness of our algorithms follow from those existing algorithms. We refer to [2, 3, 54, 55] for more detailed discussion and proof of correctness of the algorithms.

Handling a large number of predicates. When the number of predicates to be monitored is large (e.g. hundreds of thousands, as in *Social Media Analysis* application in the next section or in graph-based applications discussed in the Introduction), it is costly to maintain monitoring resources (memory, CPU cycles) for all of them simultaneously. That not only slows down the detection latency but also consumes all the resources on the machines hosting the monitors (for example, we received `OutOfMemoryError` error when monitoring tens of thousands of predicates simultaneously). However, we observe that not all predicates are active at the same time. Only predicates relevant to the nodes that the clients are currently working on are active. A predicate is considered inactive when there is no activity related to that predicate for a predetermined period of time, and therefore the evaluation of that predicate is unchanged. Hence, the monitors can clean up resources allocated for that predicate to save memory and processing time.

Automatic inference of predicate from variable names. This feature is also motivated by applications where the number of predicates to be monitored is large such as the graph-based applications. In this case, it is impossible for the users to manually specify all the predicates. However, if the variables relevant to the predicates follow some naming convention, our monitoring module can automatically generate predicates on-demand. For example, in graph-based applications, the predicates are the mutual exclusion on any edge whose endpoints are assigned to two different clients. Let A_B is such an edge, and assume $A < B$. If the clients are using Peterson's mutual

exclusion, the predicate for edge A_B will be

$$\neg\mathcal{P}_{A\_B} \equiv (flagA\_B\_A = true \wedge turnA\_B = \text{"}A\text{"}) \\ \wedge (flagA\_B\_B = true \wedge turnA\_B = \text{"}B\text{"})$$

When a server receives a request (PUT or GET) from some client for a variable whose name is either `flagA_B_A`, or `flagA_B_B`, or `turnA_B`, it knows that the client is interested in the lock for edge A_B and the server will generate the predicate for edge A_B so that the monitors can detect if the mutual exclusion access on edge A_B is violated. On the other hand, if the servers never see requests for variables `flagA_B_A`, `flagA_B_B`, and `turnA_B`, then both nodes A and B are assigned to the same client and we do not need the mutual exclusion predicate for edge A_B.

3.2 Rollback from Violations

3.2.1 Rollback Mechanism

While data anomalies are possible with eventual consistency, they are rare [4] given that networks are reliable and client conflicts are infrequent. However, such data anomalies and client conflicts can arise and, hence, one needs to deal with these anomalies if we are using an application that relies on eventual consistency. We discuss the rollback approaches for such scenarios.

One possible approach for rollback, especially if violations can be detected quickly is as follows: We partition the work assigned to each client in terms of several tasks. Each task consists of two phases (cf. Figure 3.5): (1) *Read* phase: the client obtains all necessary locks for all nodes in the task, reading the necessary data, and identify the values that need to be changed. However, all updates in this phase are done in local memory. (2) *Write* phase: the client writes the data that they are expected to change and reflect it in the data store.

In such a system, a violation could occur if clients C1 and C2 are accessing the same data simultaneously. For sake of discussion, suppose that client C1 started accessing the data before C2. Now, if the detection of violation is quick then detection would occur before client C2 enters the write phase. In this case, client C2 has not performed any changes to the key-value store. In

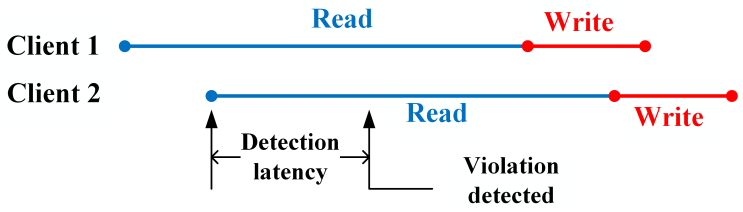


Figure 3.5: Two client tasks involved in a violation. Since detection latency is much smaller than the *Read* phase time, violation will be notified within *Read* phase of the current task of at least one client.

other words, client C2 can re-start its task (that involves reading the data from the key-value store) to recover from the violation.

With this intuition, we can provide recovery as follows: When a violation is detected, if the client causing the violation is in the read phase, it aborts that task and starts that task again. On the other hand, if a client is in write phase (and this can happen to at most one task if detection is quick enough) then it continues its task normally. Note that with this approach, it is possible that two clients that result in a violation are both in the read phase. While one of the clients could be allowed to continue normally, this requires clients to know the status of other clients. We do not consider this option as it is expected that in most applications clients do not communicate directly. Rather, they communicate only via the key-value store. We utilize this approach in our rollback mechanism. In particular, when detection is quick, we use the Algorithm 3 for rollback (cf. Figure 3.5 and Algorithm 3).

Other approaches for rollback are as follows:

- **Rollback via Retroscope [26].** The most general approach is to utilize an algorithm such as RetroScope [26]. Specifically, it allows one to rollback the state of the key-value store to an earlier state. The time, t, of rollback is chosen in such a way that there are no violations before time t. Upon such a rollback, we can determine the phases the clients are in at time t. If a client is in the read phase at time t, it will abort its current task and begin it again. And, if the client is in a write phase, it will finish that phase. Note that since there are no violations until time t, such write phases will not cause incorrect computation results.

Algorithm 3 Rollback algorithm at a client

```
1: for taskId = clientFirstTask to clientLastTask do
2:     while (performTask(taskId) == False) do
3:     end while
4: end for
5:
6: function PERFORMTASK(taskId)
7:     Obtain relevant locks
8:     Read information from data-store
9:     Compute new values
10:    if Violation is received then
11:        Release locks
12:        return False                                  ▷ abort
13:    end if
14:    Write new values to data-store
15:    return True                                       ▷ success
16: end function
```

While this approach is most general, it is also potentially expensive. Hence, some alternate approaches are as follows:

- **Use of Self-Stabilizing Algorithms.** One possibility is if we are using a self-stabilizing algorithm. An algorithm is self-stabilizing if it is guaranteed to recover to a legitimate state even in the presence of arbitrary state perturbation. This approach is discussed in Chapter 4.

- **Use of Application-Specific Rollbacks.** Another possibility is application specific rollback. To illustrate this, consider an example of graph coloring. For sake of illustration, consider that we have three nodes A, B, C, arranged in a line with node B in the middle. Each node may have additional neighbors as well. Node A chooses its color based on the colors of its neighbors. Subsequently, node B chooses its color based on node A (and other neighbors of B). Afterward, C chooses its color based on B (and other neighbors of C). At this point, node B is required to rollback, it can still choose its color based on the new color of node C while still satisfying the constraints of graph coloring. In other words, in this application, we do not need to worry about cascading rollback.

3.2.2 Dealing with Potential of Livelocks

One potential issue with rollback is a possibility of livelocks. Specifically, if two clients C1 and C2 rollback and continue their execution then the same violation is likely to happen again. We consider the following choices for dealing with such livelocks.

- **Random Backoff.** Upon rollback, clients perform a random backoff. With backoff, the requests for locks from clients arrive at different times in the key-value store. Hence, the second client is likely to observe locks obtained by the first client in a consistent manner. In turn, this will reduce the possibility of the same violation to recur.

- **Reordering of Tasks.** If the work assigned to clients consists of several independent tasks, then clients can reorder the tasks upon detecting a violation. In this case, the clients involved in the rollback are likely to access different data and, hence, the possibility of another violation is reduced.

- **Moving to Sequential Consistency.** If the number of violations is beyond a certain threshold, clients may conclude that the cost of rollback is too high and, hence, they can move to sequential consistency. While this causes one to lose the benefits of an eventual consistent key-value store, there would be no need for rollback or monitoring.

3.3 Evaluation Results

3.3.1 Experimental Setup

System configurations. We ran experiments on Amazon AWS EC2 instances. The servers ran on M5.xlarge instances with 4 vCPUs, 16 GB RAM, and a GP2 general-purpose solid-state drive storage volume. The clients ran on M5.large instances with 2 vCPUs and 8 GB RAM. The EC2 instances were located in three AWS regions: Ohio, U.S; Oregon, U.S; Frankfurt, Germany.

We also ran experiments on our local lab network which is set up so that we can control network latency. We used 9 commodity PCs, 3 for servers, 6 for clients, with configurations as in Table

3.1. Each client machine hosted multiple client processes, while each server machine hosted one Voldemort server process.

Table 3.1: Machine configuration in local lab experiments

Machine	CPU	RAM
Server machine 1, 2	4 Intel Core i5 3.33 GHz	4 GB
Server machine 3	4 Intel Core i3 3.70 GHz	8 GB
Client machine 1, 2	4 Intel Core i5 3.33 GHz	4 GB
Client machine 3, 4	Intel Core Duo 3.00 GHz	4 GB
Client machine 5	4 AMD Athlon II 2.8 GHz	6 GB
Client machine 6	4 Intel Core i5 2.30 GHz	4 GB

On the local network, we control the delay by placing proxies between the clients and the servers. For all clients on the same physical machine, there is one proxy process for those clients. All communication between those clients and any server is relayed through that proxy (cf. Figure 3.6a). Due to the proxy delays, machines are virtually arranged into three regions as in Figure 3.6b. Latency within a region is small (2 ms) while those across regions are high and tunable (e.g. 50, 100 ms). Since Voldemort uses active replication, we do not place proxies between servers. The latency in the proxies is simulated to follow the Gamma distribution [56, 57].

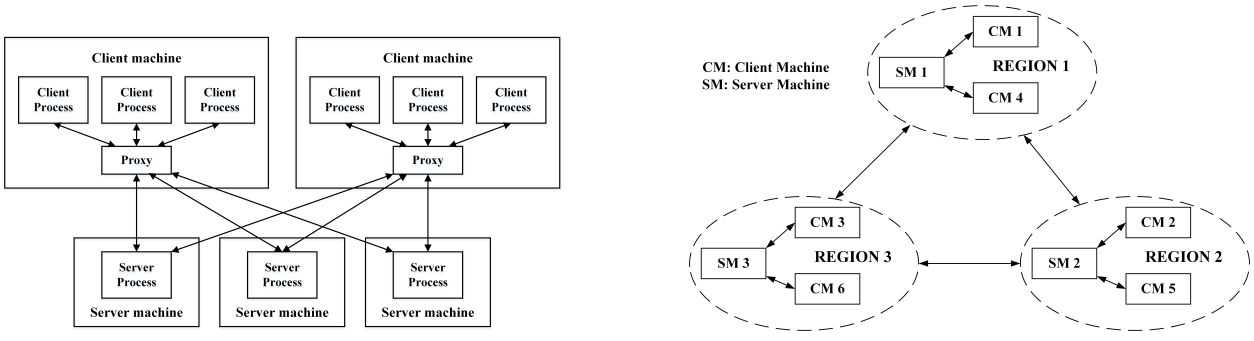


Figure 3.6: Simulating network delay using proxies. The proxies virtually partition our local lab network into three regions

We considered replication factors (N) of 3 and 5. The parameters R (required reads) and W (required writes) are chosen to achieve different consistency models as shown in Table 3.2. The

number of servers is equal to the replication factor N. The number of clients is varied between 15 and 90.

Table 3.2: Setup of consistency models with N (replication factor), R (required reads), and W (required writes)

N	R	W	Abbreviation	Consistency model
3	1	3	N3R1W3	Sequential
	2	2	N2R2W2	Sequential
	1	1	N3R1W1	Eventual
5	1	5	N5R1W5	Sequential
	3	3	N5R3W3	Sequential
	1	1	N5R1W1	Eventual

Test cases. In our experiments, we used 3 case studies: *Social Media Analysis*, *Weather Monitoring*, and *Conjunctive*.

The application motivated by *Social Media Analysis* considers a large graph representing users and their connections. The goal of clients is to update the state of each user (node) based on its connections. For the sake of analysis in our analysis, the attribute associated with each user is a color and the task is to assign each node a color that is different from its neighbors. We use the tool *networkx* [58] to generate input graphs. There are two types of graph: (1) Power-law clustering graph that simulates the power-law degree and clustering characteristics of social networks, and (2) Random 6-regular graph in which each node has 6 adjacent edges and the edges are selected randomly. The reason we use random regular graphs is that they are the test cases where the workload is distributed evenly between clients and throughout the execution. The graphs have 50,000 nodes with about 150,000 edges. Each client is assigned a set of nodes to be colored and run a distributed coloring algorithm [25].

Since the color of a node is chosen based on its neighbors' colors, while a client C_1 is coloring node v_1, no other client is updating the colors of v_1's neighbors. The goal of the monitors is to detect violation of this requirement. This requirement can be viewed as a mutual exclusion (semi-linear) predicate where a client going to update the color of v_1 has to obtain all the exclusive locks

associated with the edges incident to v_1. Mutual exclusion is guaranteed if clients use Peterson's algorithm and the system provides sequential consistency [59]. However, it may be violated in the eventual consistency model. To avoid deadlock, clients obtain locks in a consistent order. For example, let A_B and C_D are the locks associated with the edges between nodes A and B, and C and D respectively. Assume $A < B$ and $C < D$. Then lock A_B is obtained before C_D when $A < C$ or when $A = C$ and $B < D$.

The number of predicates being monitored in this test case is proportional to the number of edges.

We note that the task performed by each client (i.e., choosing the color of a node) is just used as an example. It is easily generalized for other analysis of Social Media Graph (e.g., finding clusters, collaborative learning, etc.)

The application motivated by *Weather Monitoring* task considers a planar graph (e.g. a line or a grid) where the state of each node is affected by the state of its neighbors. In a line-based graph, all the nodes of the graph are arranged on a line and each client is assigned a segment of the line. In a grid-based graph, the graph nodes are arranged on a grid. The clients are also organized as a grid and each client is responsible for a section of the grid of nodes. In this application, we model a client that updates the state of each node by reading the state of its neighbors and updating its own state. This application can be tailored to vary the ratio of GET/PUT request. This application is relevant to several practical planar graph problem such as weather forecasting [60], radio-coloring in wireless and sensor network [61], computing Voronoi diagram [62].

Finally, the *Conjunctive* application is an instance of distributed debugging where the predicate being detected (i.e., $\neg\mathcal{P}$) is of the form $\mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \cdots \wedge \mathcal{P}_l$. Each local predicate $\mathcal{P}_i$ becomes true with a probability β and the goal of the monitors is to determine if the global conjunctive predicate $\neg\mathcal{P}$ becomes true. In this application, we monitor multiple conjunctive predicates simultaneously. Since we can control how frequently these predicates become true by varying β, we can use it mainly to assess monitoring latency and stress the monitors. Conjunctive predicates are also useful in distributed testing such as to specify breakpoints.

Performance metric and measurement. We use throughput as the performance metrics in our experiments. Throughput can be measured at two perspectives: application, and Voldemort server. The two perspectives are not the same but related. One application request triggers multiple requests at Voldemort client. For example, one application PUT request is translated into one GET_VERSION request (to obtain the last version of the key) and one PUT request (with a new incremented version) at the Voldemort client library. Then each Voldemort client request causes multiple requests at servers due to replication. Failures and timeout also make the counts at the applications and the servers differ. For example, an application request is served and counted at a server but if the server response is lost or arrives after the timeout, the request is considered unsuccessful and thus not counted at the application. Generally, servers' counts are greater than applications' counts. In our experiments, we use the aggregated measurement at servers to assess the overhead of our approach since the monitors directly interfere with the operation of the server, and use aggregated measurement at applications to assess the benefit of our approach because that measurement is close to users' perspective. Hence, in the following sections, for the same experiment, we note that the measurements used for overhead and benefit evaluation are different.

Stabilization of the Results. We ran each experiment three times and used the average as the representative results for that experiment. Figure 3.7 shows the stabilization of different runs of an experiment. Note that the values are aggregated from all applications. We observe that in every run, after a short period of initialization, the measurements converge on a stable value. When evaluating our approach, we use the values measured at the stable phase. We also note that the aggregated throughput in Figure 3.7 is not very high but expected. The pairwise round-trip latency between three AWS regions (Ohio, Oregon, Frankfurt) were 76, 103, and 163 ms. The average round-trip latency was 114 ms. On M5.xlarge EC2 instances with a GP2 storage volume, the average I/O latency for a read and a write operation was roughly 0.3 ms and 0.5 ms, respectively. We will roughly estimate the cost of a GET request since in *Social Media Analysis*, most operations are GET requests to read lock availability and colors of neighbors. Assume eventual consistency

R1W1 is used, a GET request is executed by Voldemort client in two steps:

1. Perform parallel request: client simultaneously sends GET requests to all servers (N = 3) and wait for responses with a timeout of 500 ms. The wait is over when either client gets responses from all servers or the timeout expires. In this case, the client will get all responses in about 114.3 ms (114 ms for communication delay, and 0.3 ms for the read operation processing time at the server).

2. Perform serial request: client checks if it has received enough required responses. If not, it has to send addition GET requests to servers to get enough number of responses. If after the additional requests, the required number of responses is not met, the GET request is considered unsuccessful. Otherwise, the result is returned. In the current case, the number of responses received (3) is greater than the required (R = 1). Thus this step is skipped.

From this discussion, a GET request takes roughly 115 ms to complete, on average. Since GET is the dominating operation in the *Social Media Analysis* application, with 15 clients, the expected aggregated throughput is $\frac{15}{0.1143} \approx 131\ ops$. The average throughput measured in experiments was 132 ops (cf. Figure 3.7).

If we run experiments where all machines are in the same region but in different availability zones, the aggregated throughput will be higher (cf. Figure 3.9). For example, in the AWS North Virginia region, the average round-trip latency within an availability zone was about 0.5 ms, and between different availability zones was about 1.4 ms. Based on the discussion about GET request above, a GET request takes roughly 0.8ms (0.5 ms for network latency within an availability zone plus 0.3 ms for processing read request at the server). Similarly, a GET_VERSION request takes 0.8 ms. Since we are using R1W1 configuration, an actual PUT request can be satisfied by the server within the same availability zone. Thus, an actual PUT request takes roughly 1 ms (0.5 ms for network latency within an availability zone plus 0.5 ms for write operation processing time at the server). A PUT request (consisting of a GET_VERSION request and an actual PUT request) takes roughly 1.8 ms. Assume the workload consists of 50% GETs and 50% PUTs, then on average,

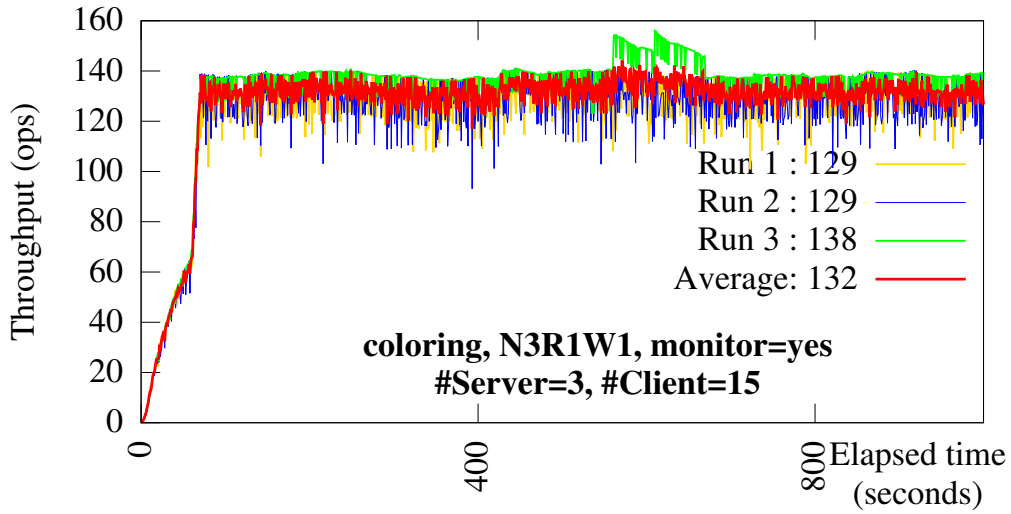
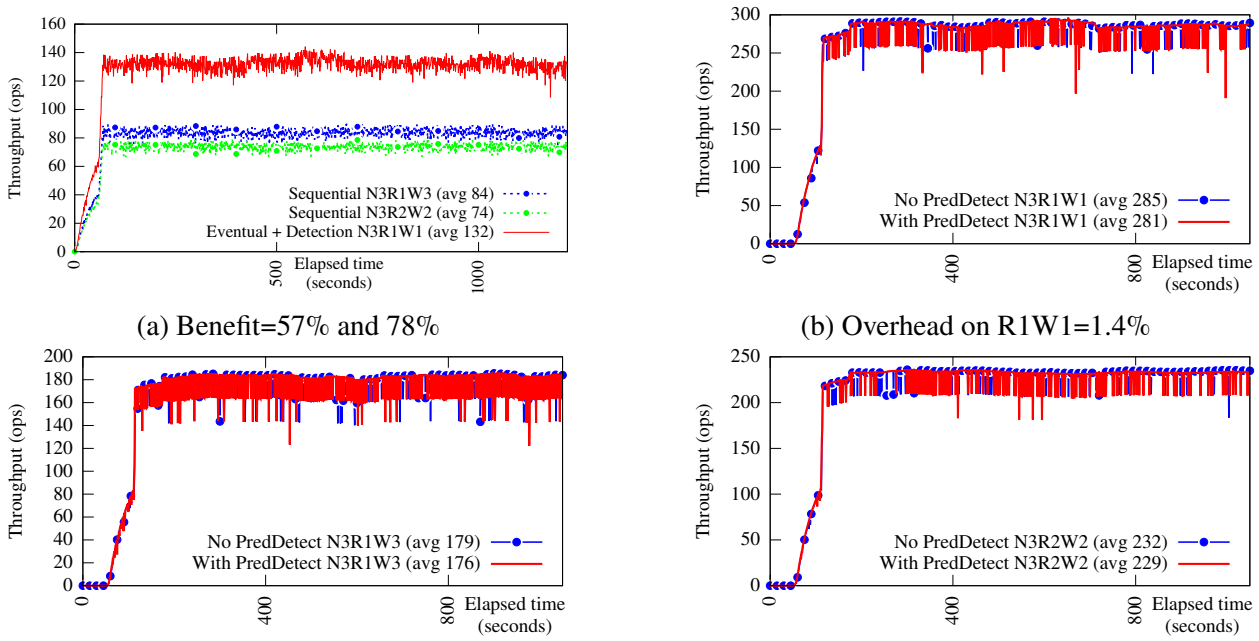


Figure 3.7: Illustration of result stabilization. The *Social Media Analysis* application is run three times on Amazon AWS with monitoring enabled. Number of servers (N) = 3. Number of clients per server (C/N) = 5. Aggregated throughput measured by *Social Media Analysis* application in three different runs and their average is shown. This average is used to represent the stable value of the application throughput.

a request takes $0.5 \times 0.8 + 0.5 \times 1.8 = 1.3\ ms = 0.0013\ s$. With 10 clients, the expected aggregate throughput is $\frac{10}{0.0013} = 7692\ ops$. If the workload consists of 75% GETs and 25% PUTs, a request takes $0.75 \times 0.8 + 0.25 \times 1.8 = 1.05\ ms = 0.00105\ s$, and the expected aggregate throughput is $\frac{10}{0.00105} = 9524\ ops$. In our experiments, the aggregate throughput measured for 25% PUT and 50% PUT was 9593 ops and 7782 ops, respectively (cf. Figure 3.9a and 3.9b).

3.3.2 Analysis of Throughput

Comparison of Eventual Consistency with Monitors vs. Sequential Consistency. As discussed in the introduction, one of the problems faced by the designers is that they have access to an algorithm that is correct under sequential consistency but the underlying key-value store provides a weaker consistency. In this case, one of the choices is to pretend as if sequential consistency is available but monitor the critical predicate $\mathcal{P}$. If this predicate is violated, we need to rollback to an earlier state and resume the computation from there. Clearly, this approach would be feasible if the monitored computation with eventual consistency provides sufficient benefit compared with sequential consistency. In this section, we evaluate this benefit.



(a) Benefit=57% and 78%

(b) Overhead on R1W1=1.4%

(c) Overhead on R1W3=1.7%

(d) Overhead on R2W2=1.3%

Figure 3.8: (AWS) *Social Media Analysis* application, 3 servers, 15 clients. The benefit of eventual consistency with monitors vs. sequential consistency without monitors (throughput improvement compared to R1W3 and R2W2 is 57% and 78%, respectively), and the overhead of running monitors on each consistency setting (the overhead is less than 2%).

Figure 3.8a compares the performance of our algorithms for eventual consistency with monitors and sequential consistency without monitors in the *Social Media Analysis* application on the AWS environment. Using our approach, the client throughput was increased by 57% (for N3R1W3) and 78% (for N3R2W2). Note that the cost of a GET request is more expensive in N3R2W2 (the required number of positive acknowledgment is 2) than in N3R1W3 (the required acknowledgment is 1). Since in the *Social Media Analysis* application GET requests dominates, the application performs better in N3R1W3 than in N3R2W2.

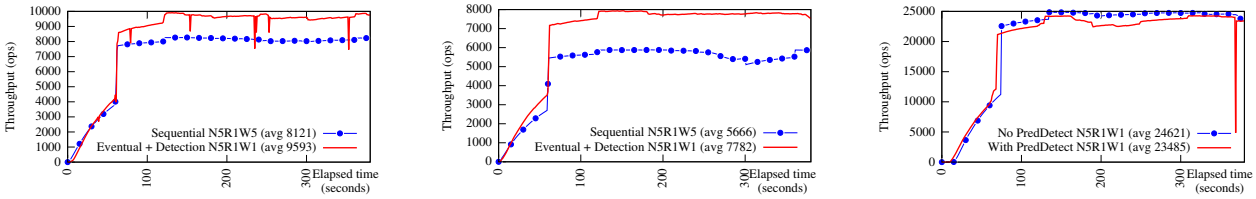
Overhead of monitoring. A weaker consistency model allows the application to increase the performance on a key-value store as illustrated above. To ensure correctness, a weaker consistency model needs monitors to detect violations and trigger rollback recovery when such violations happen. As a separate tool, the monitors are useful in debugging to ensure that the program

satisfies the desired property throughout the execution. In all cases, it is desirable that the overhead of the monitors is small so that they would not curtail the benefit of weaker consistency or make the debugging cost expensive.

Figures 3.8b 3.8c, and 3.8d show the overhead of the monitors on different consistency settings in the *Social Media Analysis* application. The overhead was between 1% and 2%. At its peak, the number of active predicates being monitored reached 20,000 predicates. Thus, the overhead remains reasonable even with monitoring many predicates simultaneously.

3.3.3 Analysis of System and Application Factors

Impact of workload characteristics. In order to evaluate the impact of workload on our algorithms we ran the *Weather Monitoring* application where the proportional of PUT and GET was configurable. The number of servers was 5 and the number of clients was 10. The machines hosting the servers and clients were in the same AWS region (North Virginia, U.S.) but in 5 different availability zones. We choose machines in the same region to reduce the latency (to less than 2 ms), thus increasing the throughput measure and stressing the servers and the monitors. If we put the clients and servers in different regions (e.g., Frankfurt Germany, Oregon USA, Ohio USA) then the throughput for 15 clients is low. To stress it further, we would have to add hundreds of clients which is very expensive. Hence, for the stress test, we put the servers and clients in the same region.



(a) PUT=25%. Benefit=18%

(b) PUT=50%. Benefit=37%

(c) PUT=50%. Overhead=4%

Figure 3.9: Benefit and overhead of monitors in *Weather Monitoring* application. Percentage of PUT requests is 25% and 50% Number of servers =5. Number of clients = 10. Machines are on the AWS North Virginia region but in different availability zones.

From Figures 3.9a and 3.9b, we find that when the percentage of PUT request increased from 25% to 50%, the benefit over sequential consistency (N5R1W5 in this case) increased from 18%

to 37%. This is because the cost for a PUT request is expensive in N5R1W5 as a PUT request is successful only when it is confirmed by all 5 servers. Thus, when the proportion of PUT increases, the performance of N5R1W5 decreases. In such cases, sequential settings that balance R and W (e.g. N5R3W3) will perform better than settings emphasize W (e.g. N5R1W5). When GET requests dominate, it is vice versa (cf. Figure 3.8a). We also observe that, when PUT percentage increased and other parameters were unchanged, the aggregated throughput measured at clients decreased. That is because a PUT request consists of a GET_VERSION request (which is as expensive as a GET request) and an actual PUT request, therefore a PUT request takes a longer time to complete than a GET request does.

Regarding overhead, Figure 3.9c shows that the overhead was 4% when PUT percentage was 50%. Note that in *Weather Monitoring* application, the number of predicates being monitored is proportional to the number of clients. Thus, the overhead remains reasonable even when monitoring several predicates simultaneously and the servers are stressed.

The number of violations detected in this experiment was only one instance in executions with a total time of $18,000\ ms$. The violation was detected within $20\ ms$.

Impact of network latency. We ran experiments on the local lab network (cf. Section 3.3.1) where the one-way latency within a region (cf. Figure 3.6b) was 1 ms and one-way latency between regions varied from $50\ ms$ to $100\ ms$. The number of clients per each server varied between 10 and 20. The values in sub-columns "server" and "app" are the aggregate throughput measured at the servers and at the applications (unit is ops). In Table 3.3, the overhead is computed by comparing server measurements when the monitors are enabled and disabled. The benefit is computed by comparing application measurements on sequential consistency without monitoring to those on eventual consistency with monitoring. For example, when one-way latency is $50\ ms$, if we run the *Weather Monitoring* application on N3R1W3, the overhead of monitoring is $(649 - 628)/649 = 3.2\%$. If we run the same application on eventual consistency N3R1W1 with monitoring, the benefit (compared to running on N3R1W3 without monitoring) is $(454 - 313)/313 = 45\%$.

Table 3.3: Overhead (oh) and benefit of monitors in local lab network. For *Conjunctive* and *Weather Monitoring*, PUT percentage is 50%.

Latency (ms)	Application	Client/ Server	Monitor	N3R1W1			N3R2W2				N3R1W3			
				server	oh	app	server	oh	app	benefit	server	oh	app	benefit
50	Conjunctive	20	yes	821	-0.2%	470	842	0.6%	375	25.3%	588	3.3%	337	40.7%
			no	819		470	847		375		608		334	
	Weather Monitoring	20	yes	924	0.2%	454	795	7.1%	345	27.2%	628	3.2%	312	45.0%
			no	926		453	856		357		649		313	
	Social Media Analysis	10	yes	560	0.2%	258	367	0.5%	156	65.4%	344	7.8%	174	47.4%
			no	561		267	369		156		373		175	
100	Conjunctive	20	yes	476	0.4%	270	491	-0.2%	218	23.3%	354	0.0%	191	42.1%
			no	478		271	490		219		354		190	
	Weather Monitoring	20	yes	544	0.7%	266	500	1.0%	209	28.5%	371	0.8%	176	49.4%
			no	548		273	505		207		374		178	
	Social Media Analysis	10	yes	287	0.0%	135	236	0.0%	74	80%	185	-0.5%	86	60.7%
			no	287		133	236		75		184		84	

From Table 3.3, as latency increases, the benefit of eventual consistency with monitoring vs. sequential consistency increases. For example, when one-way latency increased from 50 ms to 100 ms, in *Social Media Analysis* application, the benefit of eventual consistency with monitoring vs. sequential consistency R1W3 increased from 47% to 60%. In the case of R2W2, the increase was from 65% to 80%. This increase is expected because when latency increases, the chance for a request to be successful at a remote server decreases. Due to strict replication requirement of sequential consistency, the client will have to repeat the request again. On the other hand, on eventual consistency, requests are likely to be successfully served a local server and the client can continue regardless of results at remote servers. Hence, as servers are distributed in more geographically disperse locations, the benefit of eventual consistency is more noticeable. Regarding overhead, it was generally less than 4%. In all cases, the overhead was at most 8%.

3.3.4 Analysis of Violations and Detection Latency

Detection latency is the time elapsed between the violation of the predicate being monitored and the time when the monitors detect it. In our experiment with *Social Media Analysis* applications on eventual consistency (N3R1W1), in several executions of total $9,000$ seconds, we detected only 2 instances of mutual exclusion violations. Detection latency for those violations were $2,238$ ms and $2,213$ ms. So for *Social Media Analysis* application, violations could happen on eventual consistency every $4,500$ s on average.

In order to evaluate the detection latency of monitors with higher statistical reliability, we need experiments where violations are more frequent. In these experiments, the clients ran *Conjunctive* application in the same AWS configuration as *Weather Monitoring* application above. The monitors have to detect violations of conjunctive predicates of the form $\mathcal{P} = \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \cdots \mathcal{P}_{10}$. Furthermore, we can control how often these predicates become true by changing when local predicates are true. In these experiments, the rate of local predicate being true (β) was 1%, which was chosen based on the time breakdown of some MapReduce applications [63, 64]. The PUT percentage was 50%. The *Conjunctive* application is designed so that the number of predicate violations

is large and to stress the monitors. We considered both eventual consistency and sequential consistency. Table 3.4 shows detection latency distribution of more than 20,000 violations recorded in the *Conjunctive* experiments. Predicate violations are generally detected promptly. Specifically, 99.93% of violations were detected in 50 ms, 99.97% of violations were detected in 1 s. There were rare cases where detection latency was greater than ten seconds. Among all the runs, the maximum detection latency recorded was 17 seconds, the average was 8 ms.

Table 3.4: Response time in 20,647 conjunctive predicate violations

Response time (milliseconds)	Count	Percentage
< 50	20,632	99.927%
$50 - 1,000$	6	0.029%
$1,000 - 10,000$	3	0.015%
$10,000 - 17,000$	6	0.029%

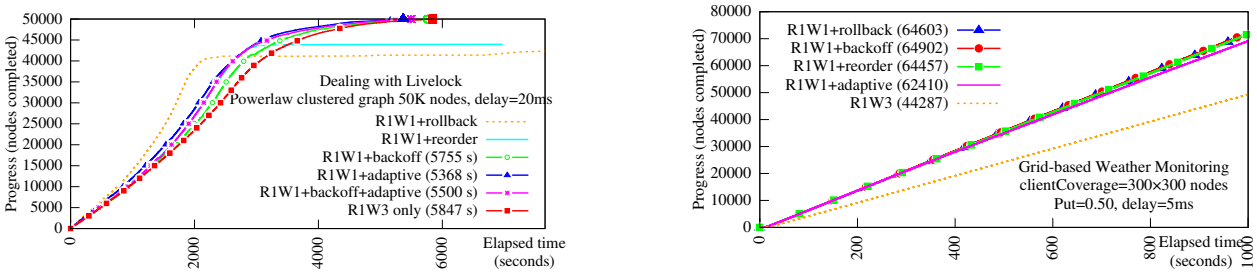
Regarding overhead and benefit, the overhead of monitors on N5R1W1, N5R1W5, and N5R3W3 was 7.81%, 6.50%, and 4.66%, respectively. The benefit of N5R1W1 over N5R1W5 and N5R3W3 was 27.90% and 20.16%, respectively.

3.3.5 Evaluating Strategies for Handling Livelocks

In this section, we evaluate the effect of rollback mechanisms. We consider the evaluation of the *Social Media Analysis* with a power-law graph and *Weather Monitoring* with grid-based graph (we describe the graphs in Section 3.3.1) . We consider the execution with sequential consistency, eventual consistency with rollback but no mechanism for dealing with livelocks, and eventual consistency with one or more mechanism for dealing with livelocks. The results are shown in Figure 3.10.

From this figure, we observe that the impact of livelocks is not the same in different applications. In particular, for terminating applications like *Social Media Analysis*, if the livelock issue is ignored, the computation does not terminate. Likewise, computation does not terminate with the mechanism of reordering of remaining tasks upon rollback. This is anticipated, in part, because recurrence of

rollback happens in end-stages where the number of remaining tasks is low. On the other hand, for a non-terminating application like *Weather Monitoring*, livelocks do not cause the computation to stall. Except for adaptive consistency, the effectiveness of different livelock handling strategies is almost similar. From Figure 3.10, we observe that rollback with adaptive consistency works best for terminating applications, and rollback with backoff works best for non-terminating applications. Therefore, we choose these mechanisms to handle livelocks in the detailed analysis of applications in Section 3.3.6.



(a) (b)

Figure 3.10: Effectiveness of livelock handling mechanisms. Number of servers=3, number of clients=30. We observed that adaptive mechanism worked best for *Social Media Analysis* (Figure 3.10a), and backoff mechanism worked best for *Weather Monitoring* (Figure 3.10b).

3.3.6 Analysis of Applications

In this section, to illustrate the benefit of our approach, we run the recovery algorithm described in Section 3.2.1 for two applications: *Weather Monitoring* and *Social Media Analysis*. We do not consider *Conjunctive*, as it was designed explicitly to cause too many violations for the purpose of detecting latency of violations. The analysis was performed in our local lab network with the round-trip latency varying between 5 ms – 50 ms. We use the approach in Section 3.3.1 to add additional delays to evaluate the behavior of the application in a realistic setting where replicas are not physically co-located. In order to deal with livelocks, we utilize the backoff mechanism for *Weather Monitoring* application, and adaptive mechanism for *Social Media Analysis* application. The number of servers was 3 and the number of clients was 30.

Weather Monitoring. When running the *Weather Monitoring* application with eventual consistency, first, we consider the nodes organized in a line. In this case, the application progressed 47.2% faster than running on sequential consistency (cf. Figure 3.11a). Even if we extend it to a grid graph, the results are similar. In Figure 3.11c, we find that in the grid graph, the application progressed 46.8% faster under eventual consistency than in sequential consistency. In both of these executions, no violations were detected in the 500 seconds and 1000 seconds window, respectively.

To evaluate the effect of rollbacks, we increase the chance of conflicts by reducing the coverage of each client (i.e. the number of nodes in the graph assigned to each client) so that the clients work on bordering nodes more frequently. In that setting, on a line graph, eventual consistency still progressed about 45% faster than running on sequential consistency (cf. Figure 3.11b), even though we had a substantial number of rollbacks (36 in 500 seconds). The detection latency for violation was on average 18 ms. The worst case detection latency was 55 ms. We note that the application motivated by *Weather Monitoring* is a non-terminating application which keeps running without termination. Hence, the number of nodes processed measured in stable phase reflects the overall progress of the application. Hence, in order to compare the progress of different experiment configurations, we measure the progress made by the clients after the same execution duration. For example, in Figure 3.11a, the larger points on each line are where we measure the progress after the execution has run for 490s. Figure 3.11b also considers the progress made by the application on eventual consistency without rollback or monitoring. Thus, the resulting answer may be incorrect. The reason for this analysis is to evaluate the cost of monitoring and rollback. As shown in Figure 3.11b, the cost of rollback is very small. Specifically, with rollback, the number of nodes processed decreased by about 1.4%.

In grid-based graphs, eventual consistency progressed 45.1% faster than sequential consistency did (cf. Figure 3.11d) even though it had to rollback a number of times (68 times in 1000 seconds). The detection latency was 10 ms on average, and 41 ms in the worst case. The cost of rollback was 1.4%.

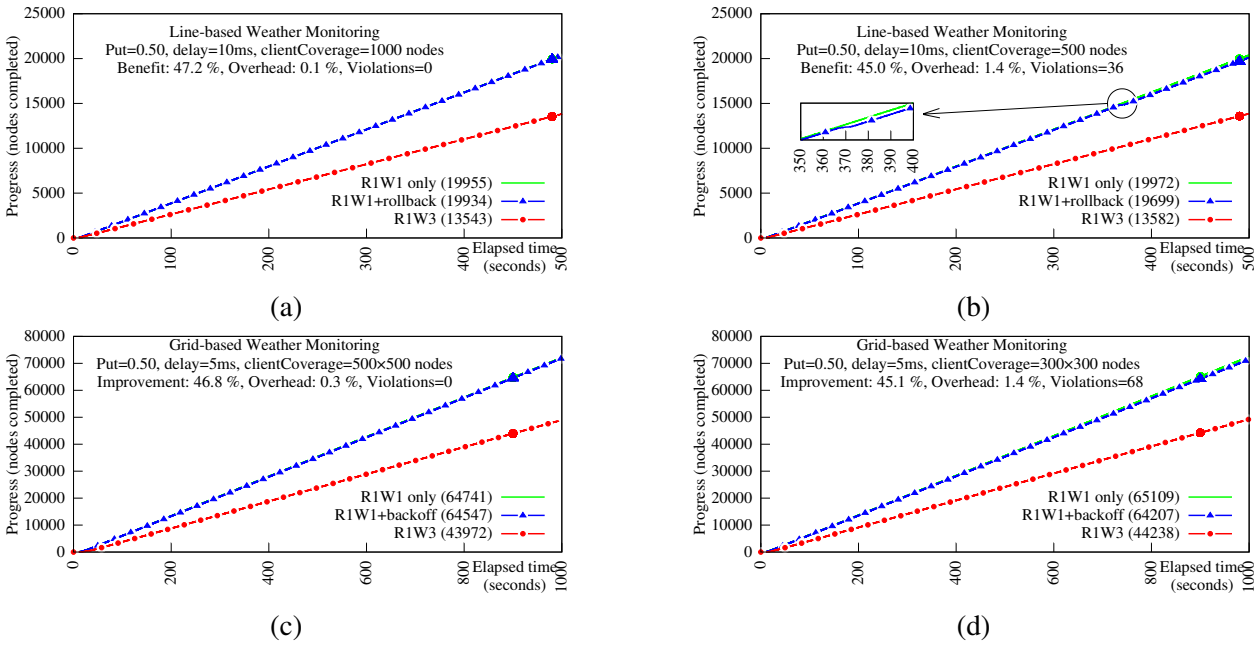


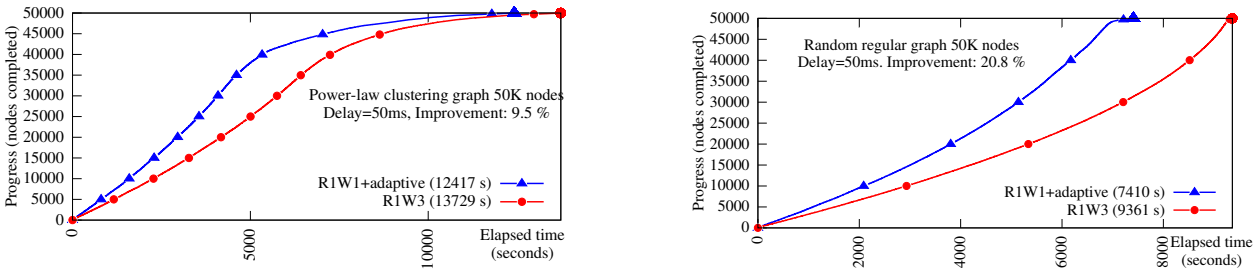
Figure 3.11: The benefit and overhead of Eventual consistency+Rollback vs. Sequential consistency in *Weather Monitoring* application. The inset figure within Figure 3.11b is a close-up view showing the impact of rollback. The larger points near the end of each data sequence are where we choose the representative values for the data sequences.

Social Media Analysis. Since the *Weather Monitoring* task is a non-terminating task, its behavior remains the same throughout the execution. Hence, to evaluate the effect of termination, we evaluate our approach in the *Social Media Analysis* application. Terminating computation suffers from the following when compared with non-terminating computations: (1) At the end, some clients may have completed their task thereby reducing the level of concurrency, and (2) The chance of rollback resulting in the same conflict increases, as the tasks remaining are very small. Hence, the computation after the rollback is more likely to be similar to the one before the rollback. In other words, the conflict is likely to recur.

We evaluate the effect of termination in two types of graph: (1) Power-law clustering (cf. Figure 3.12a), and (2) Regular graphs (cf. Figure 3.12b) where degrees of all nodes are *close*. (The details of these graphs is given in Section 3.3.1.)

On power-law clustering graphs, as shown in Figure 3.12a, before the execution reached 90% completion of the work, eventual consistency – even with the cost of monitoring and rolling back –

progressed about 18.5% faster than sequential consistency. However, in the remaining 10% of the work, when there were a few nodes to be colored, the chance of conflict increased. Furthermore, the same conflict occurred after rollback as well. Hence, in the final phase, execution under eventual consistency almost stalled due to frequent rollbacks. When the clients utilized adaptive consistency then they could make progress through the final phase and finished about 9.5% faster than sequential consistency. We note that the decline in computation rate in the final phase is also true for sequential consistency, and that is related to a property of power-law cluster graph that some nodes are high degree nodes. In regular random graph, we do not observe this decline as shown in Figure 3.12b. The main reason for this is that the likelihood of conflict in the power-law graph is high since there are several nodes with a high degree. Furthermore, it is difficult to distribute the workload of power-law clustering graph to the clients evenly. Therefore, in the final phase, some clients have completed before the others, thus reducing the parallelism. By contrast, in the regular graph, the likelihood of conflict in end stages remains the same and the workload can be evenly distributed among the clients. On a regular graph, eventual consistency with monitoring and rollback was 26% faster than sequential consistency before 90% of the nodes were processed, and 20.8% faster overall (cf. Figure 3.12b).



(a) (b)

Figure 3.12: Comparing the completion time of Sequential Consistency (R1W3) vs. Eventual Consistency with rollback and adaptive consistency (R1W1+adaptive) in *Social Media Analysis* application. On a power-law clustering graph, before 90% of the nodes are processed, R1W1+adaptive progresses about 18% faster than R1W3. Overall, R1W1+adaptive is 9.5% faster than R1W3. On a regular random graph, the benefit before 90% of the nodes are processed is 26% and the overall benefit is 20.8%.

3.4 Summary

In this chapter, we investigate the benefits of the detect-rollback approach in which we run an application on the key-value store with eventual consistency and use the monitors to detect consistency violations. Upon a violation is detected, we use our application-specific rollback algorithm to correct the application inconsistent results.

We run experiments on several test cases to quantify the benefits of the detect-rollback approach. In particular, we evaluate the overhead and effectiveness of the monitors, the benefits of running eventual consistency with monitors vs. running sequential consistency, the benefits of using detect-rollback (i.e. running eventual consistency with monitors and rollback) vs. running sequential consistency. The reason that we are interested in evaluating the benefits of eventual consistency with monitors (but without rollback) is that it informs us about the intermediate benefits of the detect phase before the rollback phase. Since the rollback phase can be done by different rollback algorithms, these intermediate benefits inform us about the efficiency of the detect phase, independent of the rollback phase.

Our experiment results show that when compared to running an application on sequential consistency, running that application on eventual consistency can improve the application throughput from 20% to 80%. There are several factors that influence the benefits such as the workload characteristics and the network latency. Specifically, when the percentage of *PUT* requests increases, the benefits increase. The reason is that the performance difference between eventual consistency and sequential consistency is larger for *PUT* requests than for *GET* requests (we explain this in Section 2.2.3). We observe that when the network latency is larger (i.e. when replicas are distributed over a large geographical area to increase fault-tolerance and availability), the benefit is higher. These observations agree with our anticipation in Section 2.2.3.

We also observe that the overhead of the monitors is low. It is typically less than 4% and in stressed experiments is less than 8%. This allows the monitors themselves to be utilized for the sake of distributed debugging. Furthermore, we observe the violations are not frequent and can be detected quickly. In particular, in a scenario designed to intentionally cause a large number of

violations, more than 99.9% of violations were detected in less than 50 milliseconds in regional networks (all clients and servers in the same Amazon AWS region), and in less than 3 seconds in global networks.

Finally, we observe that running an application with detect-rollback helps the application progress between 10%–50% faster than running on sequential consistency. An important factor that influences the benefits is re-occurring violations (livelocks). On non-terminating applications such as *Weather Monitoring*, the chance of livelocks is low and the benefits are higher. In contrast, on terminating application such as *Social Media Analysis*, the chance of livelock is high during the last phase of the computation. This requires an adaptation from eventual consistency to sequential consistency and the benefits are reduced.

In conclusion, experiment results in this chapter show that the detect-rollback approach is promising in improving the performance of computation on a key-value store. There are several possible directions to improve and extend the results of this approach. We will discuss these directions in Chapter 6.

CHAPTER 4

STABILIZATION APPROACH

In this chapter, we investigate the stabilization approach for handling cvfs. Unlike the detect-rollback approach in chapter 3, the stabilization approach does not need additional mechanisms to handle cvfs except that the existing algorithm (for sequential consistency) is stabilizing. As discussed in Section 2.4, cvfs are treated as state perturbations and a stabilizing algorithm is already designed to handle them.

We begin with Section 4.1 where we discuss the properties of cvfs and anticipate that their effect on the convergence time of a self-stabilizing program [1] is small. Next, Section 4.2 describes the termination detection algorithm which measures the convergence time of a stabilizing program. Since our focus is silently stabilizing programs, the convergence time is used as the performance metrics of evaluation and comparison. Section 4.3 presents the experimental results for the evaluation of our hypothesis. Section 4.4 discusses extension of the approach in other versions of stabilization. Section 4.5 summarizes the content of this chapter.

4.1 Expected Properties of cvf.

If we run a distributed program in the passive-node model – with a large number of nodes but relatively fewer clients– with an eventually consistent key-value store then its execution would be a computation in the presence of cvf (cf. Section 2.4). We expect the following properties for cvf:

- A single cvf only affects one node.
- cvf is expected to be rare; to be affected by cvf, we need to have one client, say $c1$, operating on node j and another client, say $c2$, operating on neighboring node k where state of j is updated on one replica but $c2$ reads it from another replica.

[1] In this dissertation, the terms self-stabilizing and stabilizing are interchangeable, and so are self-stabilization and stabilization.

- By design, cvf is not deliberate. While some **specific** single perturbation in a stabilizing program can significantly affect the convergence property, the probability that cvf would result in that specific perturbation is small.

- Between two cvf transitions, the program is likely to execute several valid transitions.

- Let $g \longrightarrow st$ be a transition of node j. One type of cvf occurs when reading an inconsistent value of some variable results in g to evaluate to false. In this case, the effect of cvf results in stuttering of the same state. In this case, the recovery of program p is unaffected.

Now, consider the execution of a program p from its arbitrary state, say s_0, in the presence of cvf. In this computation, p is attempting to change its state so that it reaches its invariant. A cvf can perturb this recovery. However, from the above discussion, the effect of cvf on recovery time is expected to be small. By contrast, the cost of eliminating cvf (i.e., utilize sequential consistency) is expected to slow down the execution of program p. In this dissertation, we evaluate this hypothesis to determine if permitting occasional cvf with eventual consistency is likely to provide us with a better recovery time that eliminating cvf with sequential consistency.

4.2 Termination Detection Algorithms.

Our termination detection algorithm to determine whether a program has reached a fixed point in the computation is based on the algorithm in [65]. We briefly describe the termination detection algorithm. Basically, the termination detector is also a Voldemort client program running a detection algorithm consisting of two rounds. In the first round, the algorithm reads the state of all nodes (including modification timestamps) and determines if every node has become disabled (i.e., all of its actions have the guards be evaluated to false). If that is true, it moves to the second round; otherwise, it restarts the first round. In the second round, the algorithm checks if the state and modification timestamp of every node is unchanged since the most recent successful first-round check. If there is any change, the algorithm restarts from the first round; otherwise, it reports the termination of computation. The termination detector runs in the consistency mode where

$R = N$ (the number of required reads equals the number of replicas) to ensure reliability. Since the termination detector only reads from and does not write to the key-value store, it minimally affects the convergence time of the computation.

4.3 Experimental Evaluation of Benefits of Stabilization in Key-Value Stores

As discussed in Section 2.4, if we run a stabilizing program with eventually consistent key-value store, it may suffer from consistency violating faults (cvf). In this section, we evaluate the hypothesis that even if the convergence is perturbed by cvf, using eventual consistency would improve the overall convergence time. We use the (silently) stabilizing algorithm by Manne et al. [40] for maximum matching to perform the evaluation. We note that our analysis depends upon the occurrence of cvf and, hence, it is equally applicable to other stabilizing algorithms as well.

4.3.1 Experiment Setup

We conduct the experiment in a local network with 9 commodity PCs (the machine configurations are described in Table 3.1). 3 PCs are reserved for the 3 key-value store servers and the clients are evenly distributed among the remaining 6 PCs.

We conduct experiments with three initial configurations: *no-match*, *random-match* and *perturbed-match*. The *no-match* experiment initializes global state so that no node is matched with any other node, characterizing execution from a properly initialized state. The *random-match* experiment initializes each node so that the match of node j is either $null$ (not matched) or some node in the network. (Of course, in the initial state if j is matched with k it does not imply that k is matched with j.) The *random-match* corresponds to a random initial state of the program. The *perturbed-match* experiment perturbs 10% of the nodes from an invariant state (i.e., a state where maximal matching has been achieved). We use the same set of initial states in each experiment. In other words, the same initial state is used to compare sequential consistency with eventual consistency. In our experiments, we use three replicas. As discussed in Section 2.2 for sequential consistency, we use R1W3 and R2W2 models whereas for eventual consistency, we use R1W1. We

repeat each experiment 3 times and take an average.

Since the maximal matching program in [40] is silently stabilizing, we use the convergence time reported by the termination detection algorithm (cf. Section 4.2) as the performance metrics of evaluation and comparison.

4.3.2 Experiment Results

We conduct five types of experiments to validate our hypothesis that permitting cvf by utilizing stabilizing algorithms is beneficial compared with the use of sequential consistency and local mutual exclusion where cvf are prevented. We conduct experiments to (1) validate this hypothesis, (2) improve performance further by improving efficiency where the occurrence of cvf is increased, and (3) evaluate the effect of concurrency (i.e., increased number of clients), (4) evaluate the convergence pattern to compare the intermediate states of the program before convergence, and (5) validate the soundness of results in a realistic environment with Amazon AWS.

Experiment 1: Sequential vs Eventual Consistency. Our first set of experiments focuses on comparing eventual consistency with cvf and sequential consistency. Recall that the latter does not suffer from cvf as each client gets the latest state of every node. The results are shown in Table 4.1.

We find that even with cvf, the convergence time with eventual consistency is significantly lower. Specifically, for configuration *no-match*, *random-match*, and *perturbed-match*, the convergence speedup factor is 1.3 – 1.7, 1.2 – 1.8, and 1.2 – 1.7, respectively. Moreover, the benefit remains fairly constant as the number of nodes increases.

Experiment 2: Revisiting Local Mutual Exclusion. Recall that to ensure that the execution in the passive-node model is free from cvf, we need to use sequential consistency and local mutual exclusion (*lme*). Note that without local mutual exclusion (*no-lme*), implementation of a protocol may suffer from inconsistencies. However, their effect is the same as cvf.

Table 4.1: Benefit of Eventual Consistency in the Presence of cvfs over Sequential Consistency. 15 Clients. With Local Mutual Exclusion. Convergence Time Unit: second.

Initial state	Consistency	Graph size (# nodes)		
		5,000	10,000	20,000
random-match	R1W1	180	399	851
	R2W2	305	637	1496
	R1W3	234	497	1080
	Speedup over R2W2	1.7	1.6	1.8
	Speedup over R1W3	1.3	1.2	1.3
perturbed-match	R1W1	123	273	650
	R2W2	184	450	1136
	R1W3	155	349	808
	Speedup over R2W2	1.5	1.6	1.7
	Speedup over R1W3	1.3	1.3	1.2
no-match	R1W1	119	273	563
	R2W2	200	445	976
	R1W3	156	363	779
	Speedup over R2W2	1.7	1.6	1.7
	Speedup over R1W3	1.3	1.3	1.4

From this observation, we first compare the convergence time for the program using local mutual exclusion and sequential consistency with the program using eventual consistency and no local mutual exclusion. The results are shown in Table 4.2.

From this table, we observe that even in the presence of increased cvf due to unavailability of local mutual exclusion (*lme*), the time for convergence is significantly lower with eventual consistency. Specifically for configurations *no-match*, *random-match*, and *perturbed-match*, the convergence speedup factor of eventual consistency without *lme* over sequential consistency (with *lme*) is 8.2 – 10.5, 7.6 – 11.6, and 7.3 – 10.3, respectively.

Experiment 3: Effect of Increased Concurrency. A key advantage of the passive-node model is that the level of concurrency can be managed. Specifically, we can increase the number of clients to increase the level of concurrency. To evaluate the effect of cvf on an increased level of concurrency, we conducted the setup for Experiment 3 with 15, 30, and 45 clients. The graph size is 10,000 nodes. The results are shown in Table 4.3. From this table, we observe that the

Table 4.2: Revisiting Local Mutual Exclusion (lme): Treating Violations as cvfs. no-lme means without local mutual exclusion. lme means with local mutual exclusion. Number of clients is 15. Convergence Time Unit: second.

Initial state	Consistency	Graph size (# nodes)		
		5,000	10,000	20,000
random-match	R1W1-no-lme	31	63	129
	R1W1-lme	180	399	851
	R2W2-lme	305	637	1496
	R1W3-lme	234	497	1080
	Speedup over R1W1-lme	5.8	6.3	6.6
	Speedup over R2W2-lme	9.9	10.0	11.6
	Speedup over R1W3-lme	7.6	7.8	8.4
perturbed-match	R1W1-no-lme	19	47	110
	R1W1-lme	123	273	650
	R2W2-lme	184	450	1136
	R1W3-lme	155	349	808
	Speedup over R1W1-lme	6.6	5.8	5.9
	Speedup over R2W2-lme	9.8	9.5	10.3
	Speedup over R1W3-lme	8.3	7.4	7.3
no-match	R1W1-no-lme	19	43	94
	R1W1-lme	119	273	563
	R2W2-lme	200	445	976
	R1W3-lme	156	363	779
	Speedup over R1W1-lme	6.2	6.3	6.0
	Speedup over R2W2-lme	10.5	10.3	10.4
	Speedup over R1W3-lme	8.2	8.4	8.3

benefit of tolerating cvfs with eventual consistency remains (fairly) same as the concurrency level is increased.

Experiment 4: Convergence pattern. The general trend of convergence for both sequential and eventual consistency looks like a sigmoid shape as shown in Figure 4.1. It starts slowly when nodes try to find their matches by making, withdrawing, and accepting proposals. Once some matches are formed, the matching progress quickly since the number of matching options is reduced. In the end, the progress slows down, as it takes time for a dead node to determine that it will remain unmatched.

Table 4.3: Effect of Increased Concurrency on the Benefit of Eventual Consistency in the Presence of $cvfs$ over Sequential Consistency. 10,000-nodes *random-match* graph. Convergence Time Unit: second

Consistency	Number of clients		
	15	30	45
R1W1-no-lme	63	52	71
R1W1-lme	399	407	500
R2W2-lme	637	638	812
R1W3-lme	497	416	548
Speedup over R1W1-lme	6.3	7.9	7.0
Speedup over R2W2-lme	10.0	12.3	11.4
Speedup over R1W3-lme	7.8	8.0	7.7

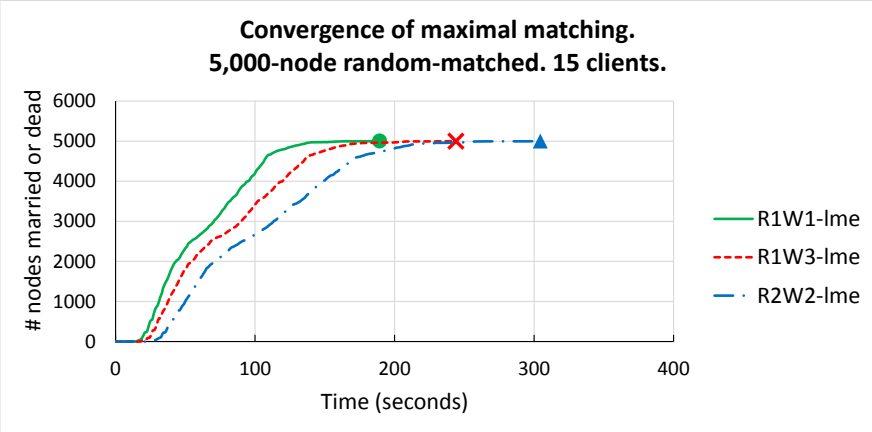


Figure 4.1: Convergence of maximal matching

From this figure, we find that at any given time t, the level of matching performed with eventual consistency is higher. In other words, the benefit of eventual consistency is not caused by the last few nodes that delay the completion of the matching algorithm.

Experiment 5: Experiments on Amazon AWS. To validate our results in a more realistic setting, we deploy similar experiments on a subset of the settings on Amazon AWS EC2 instances. The servers run on M5.xlarge instances (4 vCPUs, 16 GB RAM), the termination detector and the clients run on M5.large instances (2 vCPUs, 8 GB RAM). The instances are distributed in three different availability zones of the same region (Ohio, USA).

As shown in Table 4.4 and in Figure 4.2 , the results in the AWS experiments have similar

characteristics as those in the experiments deployed on local machines, except that it takes longer time to converge in the AWS experiments because of longer network latency. In fact, the benefit in Amazon AWS experiments is higher than the values observed in Experiment 1. This is due to the fact that latencies in Amazon AWS network are higher than in Experiment 1 where machines are on the same local network. In other words, increased latency is improving the benefit of eventual consistency with cvf over sequential consistency.

Table 4.4: AWS Experiments. Benefit of Eventual Consistency in the Presence of cvfs over Sequential Consistency. 15 Clients. Convergence Time Unit: second.

Initial state	Consistency	Graph size (# nodes)		
		5,000	10,000	20,000
random-match	R1W1-no-lme	66	139	277
	R1W1-lme	385	938	1629
	R2W2-lme	791	1666	3307
	R1W3-lme	548	1249	2426
	Speedup over R1W1-lme	5.8	6.8	5.9
	Speedup over R2W2-lme	12.0	12.0	11.9
	Speedup over R1W3-lme	8.3	9.0	8.7
perturbed-match	R1W1-no-lme	48	114	238
	R1W1-lme	250	582	1345
	R2W2-lme	531	1283	2262
	R1W3-lme	373	877	1902
	Speedup over R1W1-lme	5.2	5.1	5.7
	Speedup over R2W2-lme	11.1	11.2	9.5
	Speedup over R1W3-lme	7.8	7.7	8.0
no-match	R1W1-no-lme	45	86	145
	R1W1-lme	241	570	1154
	R2W2-lme	524	1099	2221
	R1W3-lme	396	817	1644
	Speedup over R1W1-lme	5.3	6.7	8.0
	Speedup over R2W2-lme	11.6	12.8	15.3
	Speedup over R1W3-lme	8.7	9.5	11.4

4.4 Discussion and Extensions

In this section, we consider stronger versions of stabilization and argue that they provide additional benefit in the context of tolerating cvfs with eventual consistency. Specifically, in

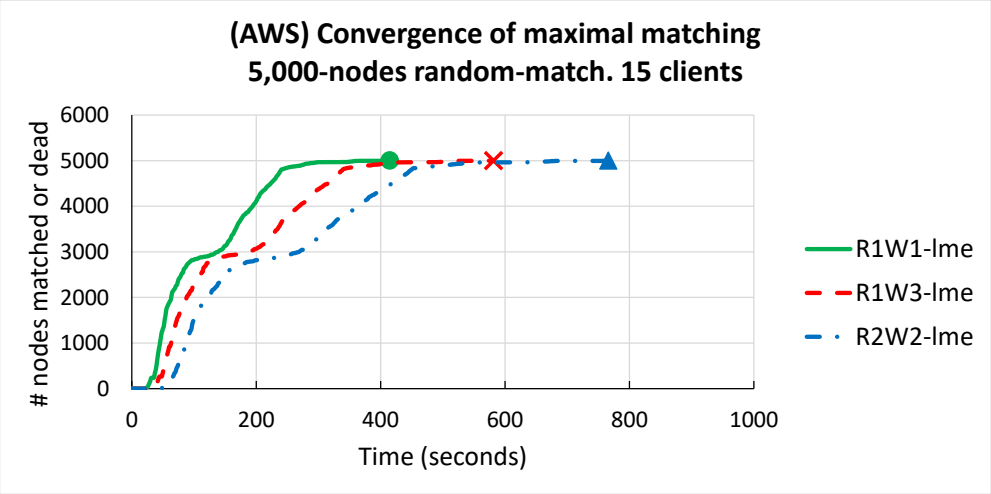


Figure 4.2: Convergence of maximal matching in the experiments deployed on Amazon EC2 instances. Note that this convergence pattern is similar to the convergence pattern in Figure 4.1 except that the convergence in Amazon EC2 experiments converges slower. This is because the delay in Amazon AWS network is longer.

Sections 4.4.1, 4.4.2 and 4.4.3, we consider benefits obtained if one begins with an active stabilizing, contained active stabilizing and fault-containment stabilizing program, respectively.

We also discuss extensions of our work. Section 4.4.4 considers the case where we use other traditional models of computations. Section 4.4.5 considers the behavior of the stabilizing program after convergence. Finally, in Section 4.4.6, we argue that stabilization is essential to achieve the benefits in Section 4.3 and Sections 4.4.1 – 4.4.3.

4.4.1 Benefits with Active Stabilization

Our analysis in Section 4.3 used experimental results to demonstrate that even in the presence of consistency violating faults (cvf), we can improve the performance of stabilizing algorithms by using eventual consistency. In this section, we show that this benefit can be formalized and enhanced if we use active stabilization from [66].

Active stabilization [66] removes a key assumption –that faults stop for a long enough time to ensure stabilization– about traditional (passive) stabilization. It was designed for cases where perturbations are caused by an adversary in the context of security.

To deal with stabilization in the presence of security related perturbations, the definition of active stabilization introduces a notion of adversary actions. Adversary actions are a (given) *subset* of $S_p \text{x} S_p$ whereas fault actions in the context of stabilization are *equal* to $S_p \text{x} S_p$, as stabilization deals recovery from an arbitrary state. We use adv_p (or adv when program p is clear from the context) to denote the adversary for program p.

When we consider computations of p in the presence of an adversary, clearly, program p must get sufficient ability to execute its actions. The definition of active stabilization from [66] uses a parameter k such that program p gets at least $k-1$ chances to execute its actions between adversary actions. Thus, the definition of computation in the presence of adversary adv_p is defined as follows:

$\langle p, adv_p, k \rangle$-**computation.** Let p be a program with state space S_p and transitions δ_p. Let adv_p be an adversary for program p. And, let k be an integer greater than 1. We say that a sequence $\langle s_0, s_1, s_2, ... \rangle$ is a $\langle p, adv_p, k \rangle$-*computation* iff

- $\forall j \geq 0 :: s_j \in S_p$, and
- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup adv_p$, and
- $\forall j \geq 0 :: ((s_j, s_{j+1}) \notin \delta_p) \;\; \Rightarrow \;\; (\forall l \mid j < l < j + k :: (s_l, s_{l+1}) \in \delta_p)$

Observe that $\langle p, adv_p, k \rangle$ computation allows execution of either program or adversary. However, once the adversary executes, for subsequent steps, if the program is able to execute (i.e., it has some action of some node whose guard is true) then some program action is executed. Only if the program has reached a state where none of its actions can execute then the adversary can execute again. After k steps, program and adversary execute non-deterministically, i.e., the adversary does not have to execute. With this notion of $\langle p, adv_p, k \rangle$-computation, we define active stabilization (from [66]) as follows:

Active stabilization. Let p be a program with state space S_p and transitions δ_p. Let adv_p be an adversary for program p, i.e., $adv \in S_p \text{x} S_p$. Let k be an integer greater than 1. We say that program p is *k-active stabilizing* with adversary adv_p for *invariant I* iff

- If we start from a state in I then execution of either a program or adversary action results in a state in I, i.e., $\forall s_0, s_1 : s_0 \in I \wedge (s_0, s_1) \in \delta_p \cup adv_p \implies s_1 \in I$
- For any sequence σ (=$\langle s_0, s_1, s_2, ... \rangle$) if σ is a $\langle p, adv, k \rangle$-computation then there exists l such that $s_l \in I$.

Although the work in [66] defines the notion of active stabilization in the context of a fixed k that is constant throughout the execution, it is possible to extend it to asymptotic value where the program is permitted to execute k steps on average between adversary steps. Now, it is straightforward to observe that cvf can be modeled as an adversary. The exact transitions of cvf can be determined upfront and the expected value of the number of steps that can be executed between cvf can be computed by experimental evaluation and/or analytical model of eventual consistency.

From the above discussion, by using active stabilization, we can precisely characterize the effect of cvf rather than rely on the *expected* properties of cvf from Section 2.4.

4.4.2 Benefits with Contained Active Stabilization

Formalizing cvf via active stabilization would allow us to provide guarantees about the effect of cvf. However, similar to passive stabilization, active stabilization requires that execution of adversary actions does not cause the program to leave its invariant. If the given program is silent stabilizing then this issue is moot, as the program state does not change in the invariant. And, at this point, cvf will not affect the state of the system, as cvf occurs when different replicas are inconsistent.

For the case, where cvfs could execute inside the invariant states, we can benefit from the use of contained active stabilization (from [66]), defined next.

Contained Active Stabilization. Let p be a program with state space S_p and transitions δ_p. Let adv_p be an adversary for p. And, let k be an integer greater than 1. We say that program p is *contained k-active stabilizing* with adversary adv_p for invariant I iff

- $\forall s_0, s_1 : s_0 \in I \wedge (s_0, s_1) \in \delta_p \;\; \Rightarrow \;\; s_1 \in I$
- For any sequence σ (=$\langle s_0, s_1, s_2, ... \rangle$) if σ is a $\langle p, adv_p, k \rangle$-computation then there exists l such that $s_l \in I$.
- For any finite sequence α (=$\langle s_0, s_1, s_2, ...s_k \rangle$) if $s_0 \in I$, $(s_0, s_1) \in adv_p$ and $(\forall j : 0 < j < k : (s_j, s_{j+1}) \in \delta_p$ then $s_k \in I$.

In the above definition, the program is guaranteed to reach the invariant even if perturbed by the adversary as long as the program can execute at least k steps between adversary actions. Moreover, even if the adversary perturbs the program outside the invariant, it recovers to the invariant before the adversary can execute again. With this approach, even if cvf occurs while the system is in the invariant, and perturbs the program outside the invariant, its correctness will be restored quickly thereby providing additional assurance about those programs.

To illustrate this property, consider the example of Dijkstra's K-value token ring program [53], where each node $j, 0 \leq j < K$ maintains a variable $x.j$. The nodes are organized in a ring. The actions of each node is as follows:

Action at node 0

$$x.0 = x.N \quad \longrightarrow \quad x.0 = (x.0 + 1) \; mod \; K$$

Action at other nodes

$$x.(j-1) \neq x.j \quad \longrightarrow \quad x.j = x.(j-1)$$

It is wellknown that $O(K)$ circulations (counted in terms of actions executed by node 0) of tokens is required to restore this program from an arbitrary state to an invariant state, where the invariant is as follows:

$$\exists j : 0 \leq j \leq N : \; (\forall k : k \leq j : x.0 = x.k) \wedge$$
$$(\forall k : k > j : x.0 = (x.k + 1) \; mod \; K)$$

Next, we consider the effect of cvf in an invariant state. To illustrate this effect, consider the case where some node, say $j \neq 0$ such that $x.j = 4$. In this case, $x.(j-1)$ is either 4 or 5.

Specifically, when $x.j$ is set to 4, $x.(j-1)$ is 4. And, subsequently, it may change to 5. In this case, except in an extreme situation discussed in the next paragraph, even if the client updating node j reads an older value, it will end up reading 4. In other words, the effect of cvf is stuttering, i.e., the program remains in an invariant state. Finally, we also note that this analysis also holds for a cvf and node 0.

In an extremely rare situation, a node may read a very old value from a replica that was offline for too long. (We can guard against it with timestamps or in systems that use passive replication where replicas synchronize periodically. But, we ignore that for now.) In this case, the client may read a random value thereby creating a scenario where we have three values in the token ring. However, the recovery time for this scenario is significantly less (at most 3 executions of node 0) than the scenario (upto K executions of node 0) where each node has a random x value.

From this discussion, it follows that in the presence of a single cvf, the recovery time is significantly faster than the scenario where the program state is arbitrary. If we ignore the extremely rare case described in the above paragraph, the token ring program is active stabilizing for the cvf under consideration. If we consider the extremely rare case, with the above analysis, we can identify the maximum time required for convergence after a single cvf. Although the details of this analysis are outside the scope of this dissertation, we can use the above discussion to find the value of k required to satisfy the constraints of the definition of contained active stabilization.

4.4.3 Benefits with Fault-Containment stabilization.

Yet another approach to address cvf is to focus on the work on fault containment. Observe that cvf, by design, affects one node. While in a stabilizing program, it is possible that corruption of one node from an invariant state may perturb the system to a state where the recovery time is very large and recovery involves all nodes in the system, fault-containment system, fault-containment stabilization focuses on eliminating this possibility.

Intuitively, fault-containment stabilization [67–71] guarantees that in addition to being stabilizing, the system guarantees that from an invariant state if only one (respectively, a small number) of

the nodes is corrupted then the convergence time is small and affects a small vicinity of the affected node(s).

In this regard, we observe that fault-containment stabilization provides spatial locality where the nodes affected by cvf would be physically close to the node that suffered from cvf. By contrast, in contained active stabilization, we get temporal locality where recovery time is small.

4.4.4 Other Traditional Models of Computation

Our model in Section 2.3.1 focused on the model that is traditionally called central daemon/interleaving semantics. Observe that the notion of cvf introduced in Section 2.4 captured the scenario where the node relied on an inconsistent value of some node to execute its action. In the model in Section 2.3.1, cvf *could result* due to a client reading the state of some node incorrectly. In other words, the notion of cvf is independent of the underlying computational model.

It follows that the notion of cvf also applies to other models such as read/write atomicity, distributed daemon, etc. Thus, having a self-stabilizing algorithm and running it with an eventually consistent key-value store would be beneficial for these programs as well.

4.4.5 Dealing with Non-Silent Algorithms

A property of maximal matching considered in Section 4.3 is that it is an instance of a silent self-stabilizing algorithm. By a silent algorithm, we mean that in a legitimate state, there are no enabled actions. (In other words, when maximal matching is performed, no node needs to execute an action). There are several problems that permit such silent solution. Examples include maximal independent set, minimal vertex cover, leader election, spanning tree construction, etc. In these algorithms, once the system reaches a legitimate state, the values of the variables remain unchanged. Hence, even with eventual consistency, no client is able to update any program variables. Our analysis is applicable to all these algorithms.

For non-silent algorithms, however, the use of eventual consistency may create certain new difficulties. We discuss them, next and identify issues in addressing them.

In a non-silent algorithm, we may be faced with a situation where we have an action, say ac (of the form $g \longrightarrow st$) that is executed by client c, that executes inside legitimate states. If we execute action ac under eventual consistency, it may be possible that g evaluates to true because c is reading an inconsistent value of the data store. In this case, execution of action ac may cause the system to be perturbed outside the legitimate states. In other words, execution of the *offending action* ac causes the system to start from a state in the invariant to a state outside the invariant. While this perturbation would (eventually) be corrected by the stabilization of the algorithm itself, this implies that with eventual consistency, execution of the algorithm from a state in the invariant may not remain within the invariant even in the absence of faults.

One approach is to utilize the notion of closure and convergence [72]. In particular, in this work, authors partition the actions of the stabilizing algorithms into *closure actions* (that execute within the invariant states) and *convergence actions* (that execute outside the invariant).

Thus, a natural question in this context is could we execute such a program so that (1) closure actions are run under sequential consistency and (2) convergence actions are run under eventual consistency. Unfortunately, this approach is incorrect. Specifically, it is possible that the program is in a state in the invariant. However, some client reads the state of some node incorrectly and thereby concludes that guard of some convergence action is true. In this case, it may execute the corresponding action. If this happens, the resulting state may be outside the invariant.

While this straightforward approach does not work for dealing with cvf for non-silent algorithms, we can use an alternative using the notion of contained-active-stabilization discussed in Section 4.4.2.

4.4.6 Non-stabilizing Algorithms and cvf

A natural question from this work is *Was it essential for the algorithm to be stabilizing to achieve the benefit identified in Sections 4.3 and 4.4.1 – 4.4.3?*

We argue that the answer to this question is *Yes*.

The reason that stabilizing programs could tolerate cvf is that, by definition, cvf is a subset

of arbitrary transient faults. Specifically, cvf corrupts the state of one node. And, a stabilizing program is designed to tolerate it. If the underlying program is not stabilizing, it is possible that the effect of even a single cvf may result in the program to reach a state where we have no knowledge about its subsequent behavior. In particular, it may cause the program to deadlock, go into a loop, etc.

Theoretically, one could benefit if the program was designed to tolerate a few cvfs that could occur at a time. However, it is possible that occurrences of multiple cvfs could affect multiple nodes at a time. Hence, we must tolerate a certain threshold t of simultaneous cvfs. However, if one follows the zero-one-infinity [73] principle of software design, unless we can argue that at most one cvf can occur at a time in the given system, we should tolerate an unbounded number of cvfs thereby *essentially* requiring the algorithm to be stabilizing.

If one must use a non-stabilizing algorithm, we can tolerate cvf as follows: Let T be a state predicate from where the program is expected to recover to its original behavior. (For stabilizing programs, T =state space. For programs that cannot tolerate even a single cvf, $T = I$, the legitimate states.) Now, we can run a monitoring algorithm for violation of T and restore the program to an earlier state if the program is perturbed outside T. A similar approach (under certain restrictions) is considered in [74]. However, this approach is limited in terms of being able to find T and being able to detect $\neg T$ efficiently at runtime.

4.5 Summary

In this chapter, we investigate the benefit of the self-stabilization approach. We observe that running self-stabilizing programs on eventual consistency helps the programs converge 1.2–1.8 times faster. Furthermore, it is interesting as we observe that we can relax the coordination mechanisms between the clients such as mutual exclusion, and treat coordination violations as cvf. Although the chance of cvf increases, the overhead of coordination mechanisms is removed. Experiment results show that the gain is actually higher than the cost. The convergence time speedup even reaches as high as 7–12 times.

We also discuss other variations of self-stabilization programs. We anticipate that eventual consistency is also beneficial in these cases. One of our future works is to validate this hypothesis. We discuss other future work related to the self-stabilization approach in Chapter 6.

CHAPTER 5

STABILIZATION VERSUS DETECT-ROLLBACK

The two previous chapters discuss the detect-rollback and stabilization approaches for handling cvfs. Experimental results show that both approaches are beneficial when compared to sequential consistency. In this chapter, we compare the benefits of the two approaches. Clearly, if the underlying program is not stabilizing then we must rely on the detect-rollback approach. Hence, we focus on stabilizing programs where both approaches are applicable. In particular, we consider three stabilizing graph computation problems/applications as our case studies: planar graph coloring, arbitrary graph coloring, and maximal matching.

The organization of this chapter is as follows. Section 5.1 explains how we set up the experiments and the performance metrics used for comparison. Section 5.2 presents the experimental results and our analysis of the effect of some factors on the performance of both approaches. More analysis is discussed in Section 5.3 to obtain further insights and implications of the results. We summarize the chapter in Section 5.4.

5.1 Experiment Setup

5.1.1 System Configuration

As in previous chapters, we ran experiments in two environments: local lab network and Amazon Web Service (AWS) network. For the configuration of the local lab network, we refer to Section 3.3.1. The configuration of the local lab machines used for the experiments in this chapter are shown in Table 5.1.

In our experiments, the distributed system consisted of 3 regions (clusters). Each region had 1 server machine (which hosted 1 replica) and 2 client machines (a client machine hosted 5 client processes). Thus, there were 3 servers and 30 clients. We chose the configuration N3R1W1 for eventual consistency, and N3R1W3 for sequential consistency (in our experiments, N3R1W3

performed better than another sequential consistency configuration N3R2W2).

On Amazon AWS platform, we used three EC2 M5.xlarge instances for the servers and six EC2 M5.large instances for the clients (cf. Table 5.1). The AWS machines are distributed in three regions (clusters): US East Ohio, US West Oregon, and Canada Central. The one-way latency among the AWS regions in our experiments (measured using `ping` command) were: US West Oregon and US East Ohio: 26 ms US West Oregon and Canada Central: 32 ms Canada Central and US East Ohio: 15 ms. The average latency between AWS regions is about 24 ms.

Table 5.1: Configurations of machines used in the experiments

Environment	Machine	CPU	RAM	Storage
Local lab	3 server machines	8 Intel Core i7-4770T 2.50 GHz	8 GB	SSD
	5 client machines	4 Intel Core i5 660 3.33 GHz	4 GB	HDD
	1 client machine	4 Intel Core i5-2500T 2.30 GHz	4 GB	HDD
AWS	3 server machines (EC2 M5.xlarge)	4 vCPUs	16 GB	SSD
	6 client machines (EC2 M5.large)	2 vCPUs	8 GB	SSD

5.1.2 Client Execution Modes.

The clients were configured to run in four different modes (cf. Table 5.2) corresponding to four different ways of executing the computation. In *sequential* mode (*SEQ*), the clients run on sequentially consistent key-value store and use mechanisms (e.g. locks) to guarantee atomicity. No cvfs occur in SEQ mode. This is the standard approach for executing the computation [75, 76] and is used as the baseline for comparison. In *eventual with stabilization* mode (*EVE-S*), the clients also employ mechanisms for atomicity but run on eventually consistent data store. This mode allows cvf to occur due to eventual consistency. However, cvf is expected to be infrequent so that between two instances of cvf, the clients can execute several transitions to stabilize the computation. *Eventual with aggressive stabilization* mode (*EVE-AS*) is similar to *EVE-S* except that the clients do not use mechanisms for atomicity. Consequently, in *EVE-AS* more cvfs are expected (Remark 2) but the locking overhead is avoided. Lastly, in *Rollback* mode, the clients run on eventually consistent data

store and also use atomicity mechanisms. Hence, cvf occurs in rollback mode. However, instead of relying on the stabilizing transitions of the program to correct cvf, the monitors are deployed to detect violations and the computation is then rolled back to undo the effect of cvf.

Table 5.2: Four client execution modes

Execution mode	Consistency	Atomicity mechanisms	Monitors	Note
SEQ	Sequential	Yes	No	No cvf. Standard approach.
EVE-S	Eventual	Yes	No	Infrequent cvf expected.
EVE-AS	Eventual	No	No	More cvf expected.
Rollback	Eventual	Yes	Yes	Rollback when violation is detected.

5.1.3 Case Study Problems

We used three stabilization problems/programs as our case studies: arbitrary/general graph coloring (*COLOR*), planar graph coloring (*P-COLOR*), and maximal matching (*MAX-MATCH*).

In *COLOR*, we have an arbitrary input graph and the goal is the assign colors to graph nodes in such a way that any two neighboring nodes have different colors. *COLOR* is a classical graph problem and has many practical applications. For example, in the problem of traffic phasing we have different traffic streams and some of them conflict with each other. We want to schedule the traffic streams in such a way that conflicting streams are not schedule at the same time. As an illustration, Figure 5.1 shows the traffic at the intersection between Wilson road and Red Cedar road near the Engineering Building. There are 12 traffic streams and some of them conflict with each other like stream 2 and stream 5. We want to schedule the green lights so that conflicting streams are not scheduled at the same time. One way to solve this problem is as follows: we model each traffic stream as a node in a graph and connect conflicting nodes by edges. By solving graph coloring problem, conflicting streams will have different colors while compatible streams could have same colors. If we schedule one time slot of green light for each color and during that time slot all traffic streams assigned that color are allowed to go, we can be sure there is no

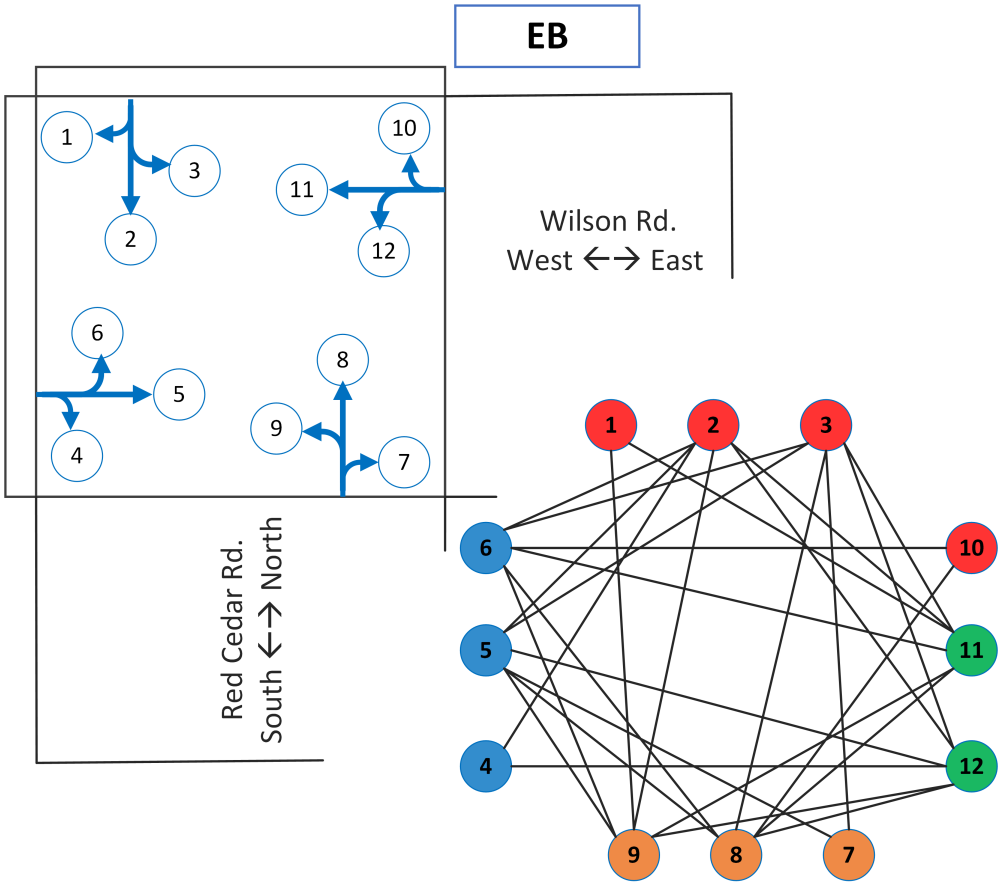


Figure 5.1: Illustration of solving a traffic phasing problem using graph coloring. Each color in the lower right graph corresponds to one time slot of green light.

traffic collision. We note that a trivial solution for *COLOR* is to assigned a different color for each node. However, such solution is not efficient. In the traffic phasing problem above, we could use 12 colors (12 green-light time slots) for 12 traffic streams, but such solution lacks concurrent flow of non-conflicting traffic streams and has significant red-light waiting time. One the other hand, coloring an arbitrary graph with minimal number of colors is known as an NP-hard problem [77]. Thus, in practice, coloring solutions that use a fewer number of colors are preferred.

By a similar approach, *COLOR* can be used in final examination scheduling (two courses having a common student are considered conflicting and their final exam schedules should not overlap so that the student could take both exams) [78], air traffic scheduling [79], task execution scheduling in GraphLab framework [76], compiler register allocation [80], bandwidth allocation [61], etc.

COLOR also has applications in banking and financial services [81], in social network analy-

sis [82] to detect community [83, 84], or recommend friends [85]. For example, one way to detect community using graph coloring is as follows. From the original graph G, we construct an overlay complete graph G_o and associate a weight for each edge in the overlay graph. The weight of an edge (v_i, v_j) in the overlay graph is defined based on the number of common neighbors between v_i and v_j in the original graph G (the higher the weight, the more connectivity between the two nodes). Then we remove from G_o those edges whose weights are greater a certain threshold and obtain a derived graph G'_o. For graph G'_o, we compute a valid coloring. Clearly, two nodes with same colors in G'_o are not neighbors in G'_o, thus the weight of the edge connecting the two nodes is higher than the threshold in G_o. In other word the two nodes have high connectivity in G and could be put in the same community. For the problem of friend recommendation, first we compute a valid coloring of the original social network (where two nodes are linked if they are friends). Observe that two nodes with the same colors are clearly not friend in the original social network and thus are potential candidates for friend recommendation (of course, other metrics such as the number of common neighbors should be considered to prioritize the recommendation).

COLOR is also used as a sub-procedure in other algorithms such as clique computation [86, 87], matrix factorization [88], graph partitioning [89]. In distributed computing, we know that if the processes run identical code and identical initial state, it is possible that the algorithm will not terminate due to symmetry [90, 91]. We could break the symmetry by assigning identifiers to processes that are locally distinguishable. Graph coloring can be used for that purpose. A valid solution of *COLOR* also naturally induces a directed acyclic graph (an edge exists from node v_i to node v_j if the numeric value corresponding to the color of node v_i is greater than that of node v_j).

For *COLOR*, we used the stabilizing algorithm in [92] (the first of three variations). One observation on this algorithm is that the result of a faulty transition, i.e. two neighboring nodes have the same color, can be corrected by just one action at one of the nodes without affecting nodes at a few hops away.

The problem of planar graph coloring is motivated by applications on planar graphs such as weather monitoring [60], radio-coloring in wireless and sensor network [61], computing Voronoi

diagram [62], etc.. We implemented the algorithm by Ghosh and Karaata [93] that guarantees to use at most 6 colors for planar graphs. This algorithm consists of two steps: constructing a directed acyclic graph (DAG) and coloring the nodes based on that DAG (in implementation, the two steps can run simultaneously).

The problem of maximal matching has many applications in resource allocation such as telephone line switching [94], college student placement [95], stable marriage [96], and matrix computation [97]. We used the algorithm in [40] to find a maximal matching of a graph. In *MAX-MATCH*, a cvf may require several actions to correct.

5.1.4 Input Graphs

We used three types of input graphs in the experiments: planar graphs, social graphs, and random regular graphs. A planar graph is a graph that can be drawn on a plane such that its edges do not cross with each other. We used the algorithm and program in [98] to generate planar graphs of approximately 10,000 nodes (and roughly 24,000 edges). In a social graph, node degrees follow the power-law distribution and nodes form clusters within the graph. In a random regular graph, nodes have the same degree and are randomly connected. We used the tool *networkx* [58] to generate social and random regular graphs. These graphs had 10,000 to 50,000 nodes.

Besides the above synthesis input graphs, we also use large-scale real-world graphs. However, due to the large size, these real-world graphs are only used for experiments in Section 5.2.4 to evaluate the scalability of our approaches.

5.1.5 Workload Partitioning Schemes.

In the passive-node model, each client is responsible for a (roughly equal) partition of the graph. We used three schemes to construct the clients' partitions. In the normal partitioning (or straight partitioning), each client is responsible for a trunk of consecutive nodes. For example, with 10,000 nodes and 10 clients, client 0 is assigned nodes 0 to 999, ..., client 9 is assigned nodes 9,000 to 9,999. In the Metis partitioning, we used graph partitioning tool Metis [99] to partition the graphs.

Metis partitioning algorithm aims to minimize the *edge-cut* partitioning objective, i.e. the number of graph edges bridging different partitions, and thus increases the locality within the partitions. In other words, it helps reduce the amount of coordination between the clients. In the random partitioning, each client is assigned a distinct set with roughly the same number of nodes randomly selected from the graph. Random partitioning distributes the workload more evenly between clients but could have negative effect on the locality of partitions.

5.1.6 Performance Metrics

Since the case study programs are silently self-stabilizing, we use convergence time (the time since the programs start until they terminate, i.e. they reach a state in the invariant) as the performance metrics. (cf. Section 4.2 for the description of termination detection algorithm.) We note that the throughput does not necessarily reflect the end-to-end performance of the programs. For example, when using lock to ensure atomicity requirement, a client is likely to wait for a lock to be available. When waiting for the lock, the client keeps sending requests to check the status of the lock. Those requests increase the throughput but do not help the program to progress.

5.2 Benefits of Stabilization versus Rollback: Comparison and Analysis

5.2.1 Stabilization vs. Rollback: Comparison and Analysis

Overall comparison. Table 5.3 shows the experiment results of running four execution modes (cf. Section 5.1.2) on different case study problems and input graphs. Input graphs are partitioned with normal partitioning scheme. We ran experiments in local lab network where the average latency was 20 ms (using proxy) and each measurement is the average of several runs. The sequential mode (*SEQ*) is used as the baseline of comparison.

In general, stabilization performed better than rollback in our case studies. Specifically, stabilization *EVE-S* improved the convergence time by 25%–35% whereas *Rollback* improved the convergence time by 29% in the best case but potentially caused the performance to suffer. Remarkably, aggressive stabilization *EVE-AS* improved the performance 2–15 times.

Table 5.3: Stabilization vs. Rollback. Graphs are partitioned in normal scheme. Network latency was 20 *ms*. *SEQ* is baseline for comparison. Rows 7-10 are convergence time benefits, shown in percentage increase or in speedup (e.g. ×5.2 means 5.2 times faster).

	Problem	Planar Graph Coloring (*P-COLOR*)	Arbitrary Graph Coloring (*COLOR*)		Maximal Matching (*MAX-MATCH*)		
	Input graph	Planar 10K	Social 50K	Regular 50K	Social 10K	Regular 10K	Planar 10K
Convergence time (seconds)	SEQ	3,887	27,995	6,518	31,581	14,859	8,545
	EVE-S	2,658	18,229	4,270	23,246	11,028	6,173
	EVE-AS	754	1,885	3,547	2,892	1,866	2,590
	Rollback	3,860	32,165	4,624	32,238	12,496	8,660
Benefit	EVE-S vs. SEQ	31.6%	34.9%	34.5%	26.4%	25.8%	27.8%
	EVE-AS vs. EVE-S	×3.5	×9.7	×1.2	×8	×5.9	×2.4
	EVE-AS vs. SEQ	×5.2	×14.9	×1.8	×10.9	×8	×3.3
	Rollback vs. SEQ	0.7%	-14.9%	29%	-2.1%	15.9%	-1.4%

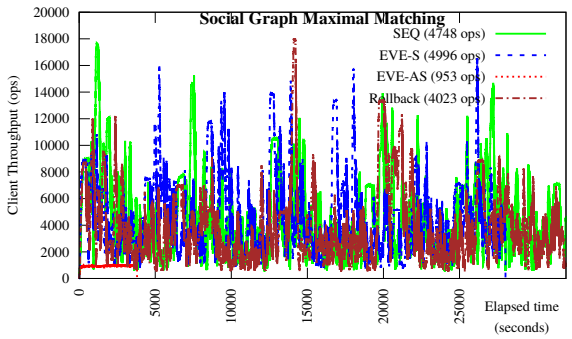
Impact of input graph structure. The structure of input graph affects the computation in two ways: (1) it changes the work balance between clients and (2) it determines the locking overhead among the clients.

In skewed graphs such as social graphs and planar graphs where there are a few nodes with very high connectivity degrees, some clients will be assigned graph partitions with more work (the number of nodes is roughly the same but the number of edges in these partitions is higher). In contrast, the workload can be evenly distributed in random regular graph due to its regularity structure.

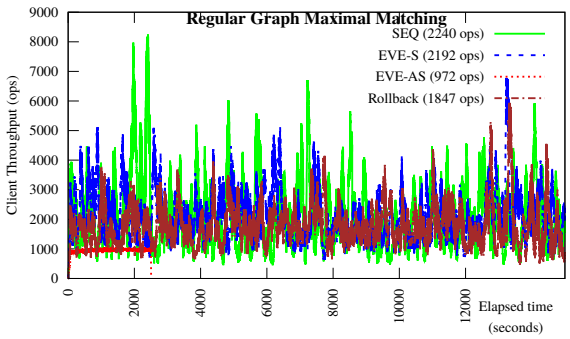
The locking overhead also depends on the connectivity structure of input graphs. For example, Figures 5.2 (a-c) measures the average throughput of *MAX-MATCH* running on 4 execution modes and in different graphs. In social graphs (cf. Figure 5.2a), the throughput in *EVE-S* (4996 *ops*) was about 5 times higher than that in *EVE-AS* (953 *ops*). In regular graphs (cf. Figure 5.2b), this difference was about 2 times (2192 and 972 *ops*). Lastly, the two throughputs were comparable in planar graph (881 and 972 *ops*, cf. Figure 5.2c). Since the key difference between *EVE-S* and *EVE-AS* is whether an atomicity mechanism is used or not (with or without locking overhead), these results indicated that the locking overhead was highest in social graphs due to their complex structure (power-law degree distribution, clustering) and lowest in planar graphs due to their locality property. (We can partition a planar graph into non-overlapping partitions with a small number of border nodes– nodes connected to other partitions.)

Due to the amount of locking overhead, computation was slowest on social graphs and (often) fastest on planar graphs. We note that *MAX-MATCH* (on *EVE-AS* mode) converged faster on random regular graphs (1,866 s) than on planar graphs (2,590 s) because some clients converged slower than others in planar graphs (due to skewed partitioning results), which increased the overall time of the program.

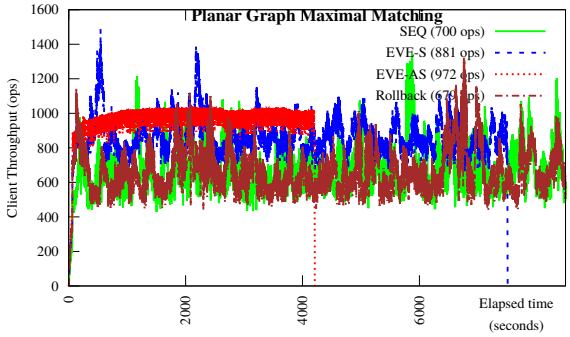
The graph structure also affects the benefits of stabilization and rollback. Specifically, the benefits of aggressive stabilization *EVE-AS* were highest (lowest, respectively) in social graphs (planar graphs, respectively) since the locking overhead was high (low, respectively) and *EVE-AS*



(a) Social graph



(b) Random Regular graph



(c) Planar Graph

Figure 5.2: Measurement of client throughput (*ops* – operations per seconds) of *MAX-MATCH* with different input graphs. Normal partitioning. Latency was 20 *ms*.

avoided such overhead. In contrast, the performance of *Rollback* suffered on social graphs because the chance of conflicts (two clients updated neighboring nodes simultaneously) was high and rollback was more frequent. *Rollback* performed well on random regular graphs since the chance of conflicts was low. We note that for *MAX-MATCH* on planar graphs, *Rollback* was slightly slower than *SEQ* because of the skewed workload. When planar graphs were partitioned using the random scheme, *Rollback* was 22% faster than *SEQ* (we discuss the impact of partitioning schemes later in this section). Finally, the benefits of *EVE-S* were fairly stable across different settings (25%–35%) because these benefits stemmed from the performance difference between eventual and sequential consistency.

Impact of case study problems. The effect of cvf is not the same for different problems. In

COLOR, a cvf can cause a client to update a node with color similar to one of its neighbors but this error can be fixed by one valid transition (update the node with a different color). Nodes that are more than one hop away are not affected by the cvf. In contrast, in *MAX-MATCH* a cvf can have cascading effect that requires updates at distant nodes. As an illustration, suppose four nodes v_1, v_2, v_3, and v_4 are on a straight line in that order. Nodes v_1 and v_3 are matched with v_2 (due to cvf). To correct this error, we can un-match v_2 from v_3. Since v_3 is now free, it can be matched with v_4, thus updating the states of both v_3 and v_4. Therefore, the cost to correct cvfs in *MAX-MATCH* is higher and the benefits of *EVE-AS* are smaller in *MAX-MATCH* than in *COLOR* ($\times$10.9 and $\times$14.9 speedup on social graphs) since *EVE-AS* introduces more cvfs.

We notice an exception: on regular graphs, the benefit of *EVE-AS* in *COLOR* is unusually low ($\times$1.8 speedup whereas the benefits of *MAX-MATCH* is $\times$8 speedup). We examined the execution of *COLOR* and found that eliminating atomicity mechanisms (in *EVE-AS*) introduced some cvfs that were difficult to recover. This happened when only a small number of nodes had inconsistent colors and cvfs caused clients to re-visit those nodes again and again. We only observed these cvfs in regular graphs as the workload was split very evenly across clients, thereby leading to a livelock. One way to address this problem is using random coloring. We will discuss our approach to overcome this issue and improve the convergence time in Section 5.2.2.

Impact of partitioning scheme. A normal partitioning of skewed graphs (social or planar) causes workload imbalance among clients and high connectivity among partitions (low locality), which increases the computation time as well as affects the benefits of stabilization and rollback. Efficient partitioning schemes can address these issues. We consider two alternatives: random partitioning and Metis partitioning. The former helps distribute the workload more evenly whereas the later improves the locality.

As shown in Table 5.4, *Rollback* was not better than *SEQ* when normal partitioning was used in *MAX-MATCH* and *P-COLOR* because of the uneven workload. However, when random partitioning was employed, the benefits of both rollback and stabilization were significantly improved (cf.

Table 5.4). However, the convergence time often increases with random partitioning because this partitioning disturbs the locality of planar graphs. To quantitatively justify this argument, we partitioned a planar graph using normal and random partitioning schemes, and measured some properties of the resultants partitions (cf. Table 5.5).

Table 5.4: Effect of random partitioning on stabilization and detect-rollback. Rows 2-5 are convergence time. Rows 6-8 are benefits, in percentage increase or in speedup (e.g. ×3 means 3 times faster). Network latency was 20 *ms*

	Execution mode	*MAX-MATCH*		*P-COLOR*	
		Normal partition	Random partition	Normal partition	Random partition
Convergence time (seconds)	SEQ	8,545	10,736	3,887	8,686
	EVE-S	6,173	7,026	2,658	5,315
	EVE-AS	2,590	1,448	754	655
	Rollback	8,660	8,341	3,860	7,242
Benefit	EVE-S vs. SEQ	27.8%	34.6%	31.6%	38.8%
	EVE-AS vs. SEQ	×3.3	×9.4	×5.2	×13.3
	Rollback vs. SEQ	-1.4%	22.3%	0.7%	16.6%

Table 5.5: Comparison between the normal and random partitioning schemes of a planar graph. For each property, the average (AVG) and standard deviation (STDEV) among the partitions are calculated.

Properties	Normal partition		Random partition	
	AVG	STDEV	AVG	STDEV
Max degree	15.3	5.0	17.1	2.7
Min degree	2.7	0.6	1.0	0.0
Total degree	1622.2	508.5	1622.2	59.9
Node count	367.8	4.7	367.8	4.7
Average degree	4.4	1.4	4.4	0.1
External edges	585.7	148.9	1568.1	54.4
Internal edges	1036.5	508.7	54.1	13.6

We observed that properties related to nodes' degrees were more evenly distributed with random partitioning. Since a node's degree reflects the cost of obtaining locks and reading state of neighbors, which constitutes a significant amount of work, an even degree distribution implies more balanced workload among partitions/clients. Consequently, we avoided slow clients that were assigned too much work. On the other hand, random partitioning potentially breaks the locality characteristics

of planar graphs. We observed with random partitioning, the number of external edges that crossed between partitions increased whereas the number of internal edges that connected nodes within a partition decreased. This implies random partitioning increased the locking overhead. Consequently, computation time often increased with random partitioning (cf. Table 5.4).

Metis partitioning [99] reduces the number of external edges bridging between partitions, and improves the locality within partitions. Consequently, the locking overhead was reduced and the convergence time of all execution modes was improved when compared to normal partitioning (cf. Table 5.6). Since the locking overhead was reduced, the benefits of aggressive stabilization *EVE-AS* decreased.

Table 5.6: Impact of Metis partitioning scheme. Latency was 20 *ms*.

Problem		*MAX-MATCH*	
Input graph		Planar	Planar
Partition scheme		Normal	Metis
Convergence time (seconds)	SEQ	8,545	2,585
	EVE-S	6,173	2,389
	EVE-AS	2,590	2,154
	Rollback	8,660	2,635
Benefit	EVE-S vs. SEQ	27.8%	7.6%
	EVE-AS vs. SEQ	×3.3	×1.2
	Rollback vs. SEQ	-1.4%	-1.9%

5.2.2 Improving the Convergence Time of Stabilization Approach

Heuristics to reduce tail latency. In the passive node model, clients are responsible for checking which nodes have enabled actions and execute those actions. The results in Table 5.3 correspond to the case where clients evaluated the guards of nodes assigned to them in a round-robin manner. One of the issues with round-robin is that some nodes whose actions are enabled may not be considered while the client is evaluating other nodes assigned to it but having no enabled actions. Note that this issue is ignored in the active node model, as, generally, it is assumed that the scheduler will choose some active node for execution. The time required for the scheduler to determine this node is ignored.

In the programs under consideration, if no action of j is enabled in the current state then this information is stable until a neighbor of j executes. Thus, if a node could tell the client that its actions are unlikely to be enabled then the client can save on reading the states of its neighbors. For such an approach to work, for node j, we need to know (1) $nd\_change.j$ the last time the client checked that no actions are enabled at j, and (2) $nbr\_change.j$ the last time one of its neighbors was updated.

Thus, when client reads the state of j and finds that $nd\_change.j > nbr\_change.j$, it does not need to read the state of its neighbors to determine if some action of j is enabled. Since clocks of all computers involved may not be identical, we change the condition to $nd\_change.j > nbr\_change.j + \Delta.j + \epsilon$ where $\Delta.j$ is the length of the last execution of j and ϵ is the upper bound for clock synchronization error. In other words, if $nd\_change.j > nbr\_change.j + \Delta.j + \epsilon$ is true then the client can save time by not issuing GET requests to neighbors of j.

Table 5.7 considers execution with this optimization. We find that this optimization is useful only when the convergence pattern exhibits a long tail at the end (cf. EVE-AS mode in Figure 5.3). The overhead of the optimization (for reading and writing additional variables) caused EVE-AS (optimized) to converge slower than EVE-AS at first. However, the optimization significantly reduced the tail of convergence graph and thus improved the overall convergence time by 44%. If the convergence pattern did not have the long tail characteristic (such as EVE-S mode in Figure 5.3, or EVE-AS mode with random coloring), this optimization increased the convergence time because of the extra overhead (cf. Table 5.7).

Randomization. As discussed in Section 5.2.1, we observed some cvfs when running *COLOR* in aggressive stabilization mode on regular graphs that prevented the program to converge. For example, suppose two clients C_1 and C_2 are working on two nodes v_1 and v_2 at the same time. Suppose the original color of both nodes is 0. Because no mutual exclusion is used in *EVE-AS*, both clients may assign the same new colors 1 for both nodes, resulting in invalid/inconsistent coloring. This error is usually resolved when one of the clients visits its node in the next round and change

Table 5.7: Effectiveness of the random coloring and the optimization for stabilization approach in the arbitrary graph coloring problem (*COLOR*). Convergence time is measured in seconds. Normal partition. Latency = 20 *ms*

Execution mode	Track update timestamp	Color selection scheme	Regular graph 50K	Social graph 50K
EVE-AS	Yes	Deterministic	1,972	4,805
EVE-AS	Yes	Random	1,941	4,807
EVE-AS	No	Deterministic	3,547	1,885
EVE-AS	No	Random	1,431	1,883
EVE-S	No	Deterministic	4,270	18,229
EVE-S	Yes	Deterministic	5,136	>20,000

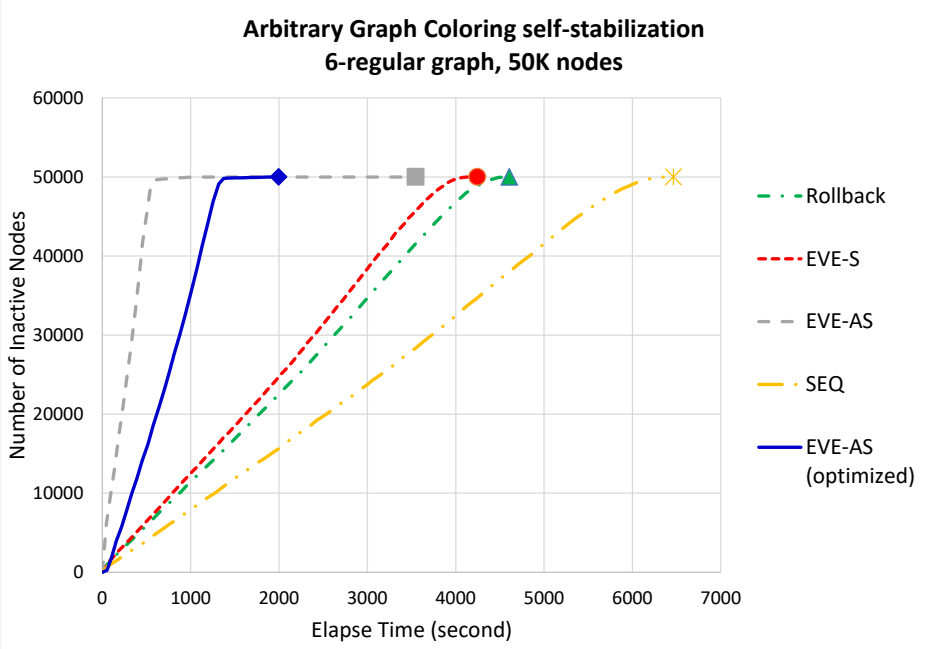


Figure 5.3: The convergence pattern of different execution modes in *COLOR*. Normal partitioning. Latency was 20 *ms*.

its node to a different color. However, if both C_1 and C_2 re-visit v_1 and v_2 at the same time, the problem persists. We observed this problem occurred only in regular graphs where the workload was split very evenly among the clients and there were only a few nodes with inconsistent colors that needed to be fixed. The problem did not happen in social graphs since the client workload was not even. In other words, running *COLOR* in *EVE-AS* mode does not guarantee convergence.

One possibility to address this problem is to modify the coloring algorithm so that the client

would choose a random value among available colors for its nodes. With this modification, *EVE-AS* is probabilistic self-stabilizing. In our experiments with the random coloring scheme, the convergence time of coloring the same regular graph in *EVE-AS* improved from 3.547 seconds to 1.431 seconds. On the other hand, the convergence time for social graph stayed almost the same (1,885 ms and 1,883 ms, cf. Table 5.7).

We also note that the performance of sequential mode (*SEQ*) is unaffected by whether deterministic coloring or random coloring is used (6,518 ms and 6,544 ms, not shown in Table 5.7). Thus, for the *COLOR* program on random regular graphs, random coloring improves the benefits of *EVE-AS* when compared to deterministic coloring (×4.6 speedup vs. ×1.8 speedup).

Impact of network latency. Network latency characterizes the geographical distribution of replicas. As shown in Table 5.8, when network latency increased (from 20 ms to 50 ms), the benefits of stabilization (*EVE-AS*) slightly increased. We attribute this result to the fact that the benefits of eventual consistency compared to sequential consistency increase when network latency increases [42]. On the other hand, the effect of network latency on the benefits of rollback was mixed. We anticipate that the different interaction patterns of rollback with the underlying stabilizing programs is the reason for this variation.

Table 5.8: Impact of network latency. Rows 4-6 are convergence time (in seconds). Rows 7-8 are the benefits, shown in percentage increase or in speedup (e.g. ×4.3 means 4.3 times faster).

Program	*MAX-MATCH*		*COLOR*		*P-COLOR*	
Input graph	Regular 10K, normal partition		Regular 10K, normal partition		Planar 10K, normal partition	
Latency	20 *ms*	50 *ms*	20 *ms*	50 *ms*	20 *ms*	50 *ms*
SEQ	14,859	35,653	6,518	15,535	3,887	9,415
EVE-AS	1,866	3,985	1,615	3,607	754	1,814
Rollback	12,496	38,657	4,742	14,113	3,860	9,057
EVE-AS vs. SEQ	×8.0	×8.9	×4.0	×4.3	×5.16	×5.19
Rollback vs. SEQ	15.9%	-8.4%	28.3%	9.2%	0.7%	3.8%

5.2.3 Experiments on Amazon AWS

To confirm the results in a more realistic deployment, we ran experiments on Amazon Web Services (AWS) network. As shown in Table 5.9, the AWS results agree with the experimental results on the local lab network (cf. Table 5.3).

Table 5.9: Experiment results on Amazon AWS network

Problem		*P-COLOR*	*COLOR*	*MAX-MATCH*
Input graph		Planar 10K	Social 10K	Regular 10K
Partition scheme		Random	Normal	Normal
Convergence time (seconds)	SEQ	10,211	21,265	6,816
	EVE-S	6,586	13,630	4,038
	EVE-AS	797	2,430	413
	Rollback	9,575	21,718	7,625
Benefit	EVE-S vs. SEQ	35.5%	35.9%	41.7%
	EVE-AS vs. SEQ	×12.8	×8.8	×16.5
	Rollback vs. SEQ	6.2%	-2.1%	-11.9%

5.2.4 Scalability Analysis

We analyze the performance of stabilization for three real-world social graphs [100]: a DBLP co-authorship network (DBLP-300K) and two YouTube friendship networks (YT-1M and YT-3M). The sizes of those graphs are provided in Table 5.10. We ran *COLOR* program on AWS machines located in three availability zones of US East Ohio region. Each availability zone has one M5.xlarge machine hosting one server/replica and two M5.large machines (each machine hosts five clients). The average one-way network latency within and between availability zones is about 0.05 and 0.3 milliseconds, respectively. The graphs are partitioned with the normal (straight) partitioning scheme.

Table 5.10 shows the experiment results. We observe that on *social* graphs, while the benefits of *EVE-S* (compared to *SEQ*) are fairly stable (25% – 30%), the benefits of *EVE-AS* increase substantially as the graph size increases. Specifically, the speedup benefit on graph with 300 thousand, 1 million, and 3 million nodes is 9.5, 13.2, and at least 59.7, respectively. This is because the performance of *EVE-AS* scales well as the graph size increases while *SEQ* does not.

We anticipate *SEQ*'s poor scalability is due to skewness in social graphs. As discussed in Section 5.2.1, it is difficult to balance workload among the clients' partitions of a social graph. As this imbalance is enlarged as a social graph grows, the computation progress is retarded on the slowest client. Moreover, a social graph contains nodes with high degree (power-law degree distribution [101]) and due to atomicity requirement (employed by *SEQ* mode), the processing time of those nodes are high. As the graph size increases, the degree of such nodes and their processing time increases quickly (cf. fifth and sixth rows in Table 5.10). Although the issue of skewness can be alleviated with complex partitioning schemes (e.g. Metis), they requires extra processing step and could not eliminate the problem. In short, as the graph size grows, skewness is more severe and the processing speed of slowest client slows down.

In contrast, the skewness problem is almost eliminated in *EVE-AS* mode since the atomicity requirement is removed and (thus) every node has roughly the same processing time (although client may need to process high degree nodes several times due to the higher chance of $cvfs$. However, as shown in Section 5.2.5, this chance is small.)

Table 5.10: (AWS) Performance of *COLOR* for different graph sizes. Normal partitioning

Input graph	DBLP-300K	YT-1M	YT-3M
Node count	317,080	1,134,890	3,223,585
Edge count	1,049,866	2,987,624	9,375,374
Average degree	6.6	5.3	5.8
Maximum degree	343	28,754	91,751
Average processing speed of slowest client (ms/node) in *SEQ* mode	156.3	389.9	2,200
SEQ time (s)	1,660	14,788	236,390
EVE-S time (s)	1,192	10,808	164,591
EVE-AS time (s)	175	1,119	2,738
Rollback time (s)	> *SEQ*	> *SEQ*	> *SEQ*
EVE-S vs. *SEQ* benefit	28.2%	26.9%	30.4%
EVE-AS vs. *SEQ* benefit	×9.5	×13.2	×86.3

5.2.5 Key Observation

Although the performance of each specific execution mode depends on several factors, in general, *EVE-AS* is noticeably efficient, *EVE-S* consistently yields substantial benefits whereas *Rollback* could provide comparable benefits as *EVE-S* but also potentially causes performance to suffer.

The above observation poses some questions: (1) what is the reason that makes stabilization, especially aggressive stabilization, more efficient than rollback? And (2) when one already has a non-stabilization algorithm for a problem at hand, and there exists another stabilizing algorithm which is (relatively) less efficient (on sequential consistency), is it worth considering the stabilizing option? This question can also be extended for handling the case where adding stabilization to a non-stabilizing program leads to an increase in overhead/computation time. We discuss these questions in Section 5.3.

5.3 Analysis of Results and Their Implications in the Design

5.3.1 Insight into Comparison of Stabilization versus Rollback

We observe from Table 5.3 that the performance of self-stabilization is generally better than rollback, particularly in *COLOR* with social graphs. We anticipate the reason is that the effect of cvfs is resolved differently in the two approaches. As an illustration, consider the *COLOR* program for social graphs. A cvf corresponds to the case when the possession time intervals of two clients for a lock overlapped. (This scenario occurs in eventual consistency when a client obtains the lock from one replica while the other client obtains that lock from another replica [51].) However, overlapping lock intervals do not necessarily mean the two clients accessed the shared data (protected by the lock) simultaneously because a client might want to obtain several locks before it started accessing the data. Furthermore, even if the clients accessed the data simultaneously, that does not necessarily mean the computed results would be wrong (the colors of neighboring nodes might still be different).

We validated this hypothesis with experiments where we ran the *COLOR* program for social graphs in *Rollback* mode. We also added instrumentation to record information about the cvfs

such as the time intervals when the clients accessed shared variables and the colors of graph nodes computed by the clients. We analyzed the recorded data after the experiments had finished and found that among 116 cvfs detected in the experiments, the client access intervals did not overlap in 35 of them. In the 81 cvfs where the clients could have accessed shared variables simultaneously, only in 6 cvfs that the computed colors were conflicting. (The reason the colors of two conflicting neighbors were still different even two clients were updating them simultaneously is that the color of a node was influenced by the colors of all of its neighbors, not just only the neighbor where the access conflict occurred.) So in most of the cvfs we observed, the conflicts were resolved favorably. These results imply that in rollback approach the program was inherently required to rollback more often than necessary (each detected cvf caused a rollback) whereas in stabilization approach the program only had to handle a few actual faulty cvfs. We believe this is one of the reasons why stabilization performed better than rollback in our experiments.

We note that the overhead of the above analysis is expensive and currently not suitable to be used with runtime rollback. It is an open problem to find efficient mechanisms to do it.

5.3.2 Results with Non-Stabilizing Algorithm

A natural question could be that what options should we choose if both stabilization and non-stabilization algorithms are available? We note that in general, when the algorithms are different, it is hard to fairly compare the two approaches since the performance is also affected by other factors such as optimization and implementation techniques. However, if the algorithms are closely similar, the comparison is useful. In this section, we also compare the stabilization and non-stabilization algorithms for graph coloring since the algorithms are fairly similar. (Our non-stabilization graph coloring algorithm is based on [25].)

Table 5.11 shows experiment results when running those algorithms on a regular random graph with 50,000 nodes, using normal partitioning scheme, in our local lab network with 20 ms latency. A key observation from this analysis is that the stabilizing algorithm is less efficient than the non-stabilizing counterpart on sequential consistency. However, it is the overall winner when used with

eventual consistency, as it can benefit from tolerating cvfs. By contrast, non-stabilizing algorithm cannot benefit from tolerating cvfs thereby resulting in lower performance even with rollback. For example, for $d = 10$ (d is the average node degree), time taken by the non-stabilizing algorithm was 7,021 s in sequential consistency and it improved to 5,456 s with eventual consistency and rollback. By contrast, the cost of the stabilizing algorithm under sequential consistency was 11,146 s. It improved to 1,717 s under EVE-AS model.

This implies that while there may be some cost associated with making the protocol stabilizing, it is recovered by tolerating cvfs. In this context, we also want to remind the reader that non-stabilizing algorithms cannot ignore cvfs, as a cvf may perturb the program to a state from where recovery is not guaranteed. Only stabilizing programs can choose to ignore cvfs as they are designed to recover from them. Non-stabilizing programs can only use the detect-rollback approach to deal with cvfs.

Table 5.11: Computation time (in seconds) of Stabilizing and Non-Stabilizing algorithms for graph coloring. The average of node degree (d) varies between 2 and 10. The baseline for calculating benefit is *SEQ*

Average node degree		d=2	d=3	d=6	d=10
Stabilizing graph coloring	SEQ	2,325	3,378	6,518	11,146
	Rollback	1,559	2,279	4,742.3	10,150
	EVE-AS	1,321	1,376	1,615	1,717
	EVE-AS benefit	×1.8	×2.5	×4.0	×6.5
Non-stabilizing graph coloring	SEQ	1,653	2,291	4,246	7,021
	Rollback	1,213	1,681	3,192	5,456
	Rollback benefit	26.6%	26.6%	24.8%	22.3%

5.4 Summary

In this chapter, we consider the detect-rollback and stabilization approaches to handle consistency violating faults (cvfs) and reduce the time for execution of stabilizing graph algorithms. Our analysis shows that for stabilizing programs, the stabilization approach provides substantial benefits compared with the detect-rollback approach. Specifically, the second approach provides a 25%–35% improvement for different programs. Furthermore, the aggressive stabilization (that

introduces additional cvfs at the cost of efficiency) reduces the convergence time 2–15 times. By contrast, the detect-rollback approach provides limited benefits and potentially causes performance to suffer when compared with sequential consistency.

We also considered another approach to reduce the time for execution. It relied on heuristics to allow clients to keep track of nodes that may have enabled actions. Experimental results show that the heuristics can improve convergence time by about 44% by reducing tail latency where the state of a very few nodes is inconsistent. However, it is only suitable for convergence patterns with a long tail of slow progress at the end.

We also find that the stabilization approach can benefit even more if the program can use other techniques to reduce overall time. Specifically, we considered the use of the random graph partitioning scheme to balance client workload. In this case, both approaches showed benefits but the benefit of stabilization was higher.

Another key insight in this work is that the benefits that apply to stabilizing algorithms can make them attractive in eventually consistent data stores even if they are (relatively) inefficient under sequential consistency. For example, in Section 5.3.2, we showed that under sequential consistency, the stabilizing program had 58% lower performance than a similar non-stabilizing program (11,146 s to 7,072 s). However, its performance was 3.2 times better under eventual consistency (1,717 s vs 5,456 s). This happened because the non-stabilizing algorithm could not tolerate cvfs in the same manner that a stabilizing program could. This indicates that there may be a substantial benefit in revising an existing algorithm for the problem at hand to make it stabilizing and reduce the overall runtime under eventual consistency.

Directions to extend the work in this chapter will be discussed in Chapter 6.

CHAPTER 6

FUTURE WORK

In this chapter, we discuss some potential directions to extend the work of this dissertation. Section 6.1 focuses on improving the detect-rollback approach. The stabilization approach is discussed in Section 6.2. Section 6.3 discusses some interesting problems that are also related to this dissertation.

6.1 Improving The Detect-Rollback Approach

Rollback with Retroscope. The rollback algorithm proposed in this dissertation is specific for some graph-based applications and has the assumption of small detection latency. It does not work for a general application in a key-value store. When the detection latency is prolonged because of network congestion or delay, the rollback algorithm also does not work because the violation occurred during a task in the past, not the current task. In this case, we have to rollback the system to a point before the task in which the violation occurred.

For the general rollback scenario, we are investigating the possibility of integrating the monitor with Retroscope [26] to automate the rollback and recovery.

Improving the monitors. The monitors used in this work suffer from false positives, i.e., they initiate rollback when it was not absolutely necessary. One possible reason for false positives is that the clients, say $C1$ and $C2$, involved in rollback had only read from the key-value store. In this case, one of the clients can continue the execution without rollback. However, in our implementation, as each client rolls back independently, both of them rollback. If this is prevented, it can not only reduce the wasted work, it can also potentially avoid the re-occurrence of conflict between $C1$ and $C2$ after rollback.

Another reason for false positives is the mismatch in the clock synchronization assumptions made by the monitors and the applications [102]. There is also the mismatch between the time

interval during which a client possesses a lock and the time interval when the client actually accesses the variables guarded by the lock. Often, the later time interval is much smaller than the former one because the client has to obtain all the locks related to a node before updating that node. This leads to the situation where the lock possession time intervals of two clients overlap but their access time intervals do not (as discussed in Section 5.3.1). Consequently, the monitors trigger false alarms and unnecessary rollbacks. In order to reduce or eliminate the false positives, we would have to augment the clients and servers with more information so that monitors will have more accurate information about the events. Such instrumentation increases the cost of monitoring but reduces the need for performing rollback.

Adaptive Consistency. Currently, the adaptive solution switches from eventual consistency to sequential consistency based on the feedback from monitors. It is possible that the increase in violations is temporary due to network issues. When the condition is resumed to normal, it would be beneficial to run in eventual consistency again. However, in sequential consistency, monitors are not required and, hence, there is no feedback mechanism to determine when using eventual consistency is reasonable. One needs to develop new techniques to permit this possibility. One possibility is the probation method. Assume a client is in sequential consistency and it wants to determine whether it is safe to switch to eventual consistency. The client waits for some predefined amount of time T_{wait} (the value of T_{wait} is determined from the analysis of experiment results) and tries eventual consistency. If violations recur quickly, it switches back to sequential consistency and doubles the wait time before the next try. The process is repeated until the client does not observe recurring violations.

6.2 Improving The Stabilization Approach

Analytical model for benefits of self-stabilizing eventual consistency. In Chapter 4, we characterize violations occurred in eventual consistency as consistency violating faults (cvf). Our experiment results show that some self-stabilizing programs could convergence in the presence

of cvf. However, it is not clear whether a designer can have an estimate of the benefits without running experiments. We are working to develop an analytical model for estimating the expected convergence speedup of self-stabilizing programs in the presence of cvf.

Other variations of stabilization As self-stabilization characteristics of different problems are different, we plan to investigate the benefit of eventual consistency in other self-stabilizing programs such as coloring [92], Dijkstra's token ring [53], and collateral composite self-stabilizing algorithms [103]. We are also interested in the benefits of eventual consistency to other variations of stabilization such as weak, probabilistic, active, and fault-containment stabilization.

Adding stabilization to non-stabilizing programs

Results in Chapter 5 show that stabilizing programs, while relatively inefficient under sequential consistency, are attractive under eventual consistency. This indicates that there may be a substantial benefit in revising an existing algorithm for the problem at hand to make it stabilizing and reduce the overall runtime under eventual consistency. We note that there are several algorithms to *add* stabilization to a non-stabilizing program [104]. These algorithms could be used in this context. However, an approach that optimizes the addition of stabilization using specific insight into the problem at hand may be more desirable as it is likely to provide the most benefit.

As another demonstration, consider the task of analyzing large-scale real-life networks (e.g. social networks) which are challenging to deal with. One of the challenges is that their complex structure imposes a significant locking coordination overhead for atomicity assurance, which retards the overall performance. Most of the existing work tried to reduce this overhead by efficient partitioning schemes [75] but the improvement was limited due to the inherent complex graph structure and required a preprocessing step. In chapter 5, we observed that aggressive stabilization (*EVE-AS*) performed particularly well in social graphs (an order of magnitude improvement) without the additional preprocessing overhead. This observation suggests that eventual consistency and stabilization is a promising candidate to efficiently tackle the complexity in social networks.

From the analysis of this work, we find that the stabilization-based approach provides a substantial benefit compared with the rollback-based approach. However, in both cases, the time required for convergence of the last few nodes is still quite high. One of the future work in this area is to reduce this overhead. Another future work is to generalize the results in chapter 5 specifically to determine which options one should choose if both stabilizing and non-stabilizing algorithms are available. Another question for investigation is whether the analysis holds for other models of distributed computation.

6.3 Other Possibilities of Future Work

6.3.1 Characteristics of Monitoring Errors

When monitoring a partially synchronized distributed system, the monitors need to know about the target system clock drift. However, this information is not always available to the monitors. Even when the target system specification is available, implementation bugs can make the actual drift differ from the specification. Therefore, the monitor assumption on clock drift may slightly differ from the application assumption. Due to the impedance mismatch between the applications and monitors about their assumptions of synchronization error, the monitors can suffer from false positives, false negatives, or both. False positives are phantom violations reported by the monitors but they did not actually occur during the execution. False positives are introduced when the monitor assumption is larger than the application assumption. False positives are false alarms and can cause unnecessary correction actions. False negatives are circumstances where the monitors failed to detect valid violations that actually occurred during the computation. They occur when the monitor assumption is smaller than the application assumption. If false negatives occur, the monitors miss to detect valid violations and the correctness of the computation is affected.

False positive rate and false negative rate are related to the precision and recall properties of the monitors. Specifically, false positive rate = 1 - precision, and false negative rate = 1 - recall. In the context where we use the monitors to detect anomalies in eventual consistency, false positives will cause unnecessary rollbacks and impede the overall progress of the computation. On the other

hand, false negatives will cause the clients to use inconsistent data and thus the computation results are compromised.

The precision and recall characteristics of the monitors is an independent problem of interest. It is useful not only in the context of recovery from violations (the detect-rollback approach) but also for distributed monitoring and debugging in general since it helps the designer use the monitors more effectively. Hence, we plan to study this problem in the future.

CHAPTER 7

RELATED WORK

In this chapter, we review the literature related to the work of this dissertation. The Voldemort key-value store and the passive-node model are related to frameworks for distributed data processing (Section 7.1). The sequential and eventual consistency are two consistency models widely supported by many distributed data stores (Section 7.2). We develop the techniques used by the detect-rollback approach based on existing work in predicate detection in distributed systems (Section 7.3), distributed snapshots (Section 7.4), and monitoring services on the cloud (Section 7.5). The approach of using stabilization to handle cvfs belongs to the literature on self-stabilization (Section 7.6). Section 7.7 summarizes the contribution of this dissertation.

7.1 Distributed Data Processing

MapReduce [105], DataFlow [106] are general-purpose distributed data processing frameworks. In the realm of distributed graph processing, many frameworks are available such as Pregel [107], GraphLab [76], GraphX [108], and PowerGraph [109]. In those works, data is persisted in semi-structural storages such Google File System, Hadoop Distributed File Systems [110], BigTable [111], or in in-memory storage such as Spark [112]. Our work focuses on the no-structure key-value stores and the impact of different consistency models on key-value store performance. Our approach's usefulness is also not limited to graph applications.

7.2 Consistency in Distributed Data Stores

Sequential consistency [12] is one of the earliest consistency models studied. It allows programmers to develop correct programs without worrying about the replication nature of a distributed system. However, for large-scale and busy systems, this consistency model is too strict that its performance does not meet the customer expectations. Weaker consistency models such as causal consistency [113, 114], FIFO consistency [1] are introduced to improve the performance while

still ensure that application-specific consistency requirements are met. At one extreme, eventual consistency model even does not provide any guarantee except a best effort. Nevertheless, eventual consistency works very well in practice and is supported by many NoSQL data stores [4, 5, 7, 10, 18, 21]. The eventual consistency model is especially popular among key-value and column-family databases. The original Dynamo [4] was one of the pioneers in the eventual consistency movement and served as the basis for Voldemort key-value store. Dynamo introduced the idea of hash-ring for data-sharding and distribution, but unlike Voldemort it relied on server-side replication instead of active client replication. Certain modern databases, such as Cosmos DB and DynamoDB [7, 8] offer tunable consistency guarantees, allowing operators to balance consistency and performance. This flexibility would enable some applications to take advantage of optimistic execution while allowing other applications to operate under stronger guarantees if needed. However, many data-stores [115, 116] are designed to provide strong consistency and may not benefit from optimistic execution module.

Aside from general-purpose databases, a variety of specialized solutions exist. For instance, TAO [117] handles social graph data at Facebook. TAO is not strongly consistent, as its main goal is performance and high scalability, even across data centers and geographical regions. Gorilla [118] is another Facebook's specialized store. It operates on performance time-series data and is highly tuned for Facebook's global architecture. Gorilla also favors availability over consistency in regards to the CAP theorem.

This dissertation focuses on eventual consistency and sequential consistency as they are available in most data stores.

7.3 Predicate Detection in Distributed Systems

Predicate detection is an important task in distributed debugging. An algorithm for capturing consistent global snapshots and detecting stable predicates was proposed by Chandy and Lamport [119]. A framework for general predicate detection is introduced by Marzullo and Neiger [49] for asynchronous systems, and Stollers [50] for partially synchronous systems. These general

frameworks face the challenge of state explosion as the predicate detection problem is NP-hard in general [3]. However, there exist efficient detection algorithms for several classes of practical predicates such as unstable predicates [55, 120, 121], conjunctive predicates [2, 122], linear predicates, semilinear predicates, bounded sum predicates [3]. Some techniques such as partial-order method [123] and computation slicing [124, 125] are also the approaches to address the NP-Completeness of predicate detection. Those works use vector clocks to determine causality and the monitors receive states directly from the constituent processes. Furthermore, the processes are static. [126, 127] address the predicate detection in dynamic distributed systems. However, the class of predicate is limited to the conjunctive predicate. In this dissertation, our algorithms are adapted for detecting the predicate from only the states of the servers in the key-value store, not from the clients. The servers are static (except failure), but the clients can be dynamics. The predicates supported include linear (including conjunctive) predicates and semilinear predicates.

In [128, 129], the monitors use Hybrid Logical Clock (HLC) to determine causality between events in a distributed execution. HLC has the advantage of low overhead but suffers from false negatives (some valid violations are not detected). In contrast, we use hybrid vector clocks to determine causality in our algorithms. In [102], the authors discussed the impact of various factors, among which is clock synchronization error, on the precision of the monitors. In this dissertation, we set ϵ at a safe upper bound for practical clock synchronization error to avoid missing potential violations. In other words, a hybrid vector clock is practically a vector clock. Furthermore, this dissertation focuses on the efficiency and effectiveness of the monitors.

7.4 Distributed Snapshots and Reset

The problem of acquiring past snapshots of a system state and rolling back to these snapshots has been studied extensively. Freeze-frame file system [130] uses Hybrid Logical Clock (HLC) to implement a multi-version Apache HDFS. Retroscope [26] takes advantage of HLC to find consistent cuts in the system's global state by examining the state-history logs independently on each node of the system. The snapshots produced by Retroscope can later be used for node reset

by a simple swapping of data-files. Eidetic systems [131] take a different approach and do not record all prior state changes. Instead, the Eidetic system records any non-deterministic changes at the operating system level and constructing a model to navigate deterministic state mutations. This allows the system to revert the state of an entire machine, including the operating system, data, and applications, to some prior point. Certain applications may not require past snapshots and instead, need to quickly identify consistent snapshots in the presence of concurrent requests affecting the data. VLS [132] is one such example designed to provide snapshots for data-analytics applications while supporting high throughput of requests executing against the system.

The overall framework for detect-rollback in this dissertation rely on the capability of efficiently taking snapshots and restoring the state of the system to a proper checkpoint. One of the directions to extend our current work is to combine our predicate detection module with a snapshot tool to rollback a general application once cvfs occur.

7.5 Monitoring Large-scale Web-services and Cloud Computing Systems.

Dapper [133] is Google's production distributed systems tracing infrastructure. The primary application for Dapper is performance monitoring to identify the sources of latency tails at scale. Making the system scalable and reducing performance overhead was facilitated by the use of adaptive sampling. The Dapper team found that a sample of just one out of thousands of requests provides sufficient information for many common uses of the tracing data.

Facebook's Mystery Machine [134] has goals similar to Google's Dapper. Both use similar methods, however mystery machine tries to accomplish the task relying on less instrumentation than Google Dapper. The novelty of the mystery machine work is that it tries to infer the component call graph implicitly via mining the logs, whereas Google Dapper instrumented each call in a meticulous manner and explicitly obtained the entire call graph.

Our predicate detection module does not rely on sampling for detection of cvfs since missing a cvf (false negative) could compromise the computation results. We formalize violations of cvfs as conjunctive and semi-conjunctive predicates and use algorithms in the literature [2, 3] to detect

them. To help our monitors scale with hundreds of thousands of predicates in a large system, we classify predicates into active and inactive ones. By keeping only active predicates in the cache, the performance of the monitors is improved (cf. Section 3.1.3).

7.6 Self Stabilization

Self-stabilization is the design principle proposed by Dijkstra [135] to help an application recover from transient faults by itself. Examples of such applications include spanning trees [27,28] leader election [29, 30], matching [31, 32], dominating set [33, 34], independent set [35, 36], clustering [37–39]. In these work, the active-node model is employed. In contrast, our work relies on the passive-node model.

A self-stabilization program has the properties of convergence and closure [72]. A self-stabilization solution does not exist for all problems [104] and weaker variations of stabilization are introduced such as weak stabilization [136], probabilistic stabilization [137]. On the other hand, stronger versions of stabilization are proposed to reduce the effect of transient faults such as active stabilization [66], collaborative stabilization [138], and fault-containment stabilization [139–143]. In this dissertation, we discuss and anticipate the benefit of eventual consistency in some versions of stabilization. It is our future work to develop an analytical model for the benefit of eventual consistency in self-stabilizing programs.

7.7 Summary

To the best of our knowledge, our work is the first one that examines and formalizes the notion of consistency violating faults (cvfs) that occur during the execution of a distributed program on an eventually consistent key-value store. We also quantitatively evaluate two optimistic execution approaches, namely detect-rollback and stabilization, that allow a computation to be run on eventual consistency and correct the cvf faults when they occur. Our results show that those optimistic execution approaches are more beneficial than to conservatively prevent cvfs with sequential consistency.

CHAPTER 8

CONCLUSION

When developing distributed applications on a key-value store, the sequential consistency model is more natural and easier for programmers to work with since it masks the complexity of replication. However, it could suffer from poor performance, especially on large-scale and busy applications. Eventual consistency provides fast performance meeting the needs of those applications but potentially suffers from data anomalies (denoted as consistency violating faults – cvfs) which compromise the correctness of the computation. In this dissertation, we investigated optimistic solutions that take the performance advantage of eventual consistency while being cvf-tolerant to ensure the correctness of the computation. Specifically, we proposed two approaches, detect-rollback and stabilization, and evaluated their benefits when compared to the standard approach of using sequential consistency.

In the detect-rollback approach, we run the application on eventual consistency and formulating cvfs as a safety predicate Φ. We use monitors to detect the existence of $\neg\Phi$ (or violations of Φ). When a violation is detected, the computation is rolled back to a point where the cvf has not occurred and the computation is resumed. Two main components of the detect-rollback approach are the monitors and the rollback mechanism. Our monitors implement the predicate detection algorithms in [2,3] with adaptation for key-value stores. For rollback mechanisms, we proposed an efficient rollback algorithm for graph-based applications. When performing rollback, we observed some challenges with livelocks in which the violations are likely to re-occur after rollback. We proposed some strategies to handle livelocks such as random back-off, adaptive consistency, and re-ordering the tasks.

Our experiment results with several graph-based applications on Amazon AWS platform as well as on our local lab network show that the detect-rollback provides performance benefits when compared to the baseline approach of running the applications on sequential consistency. Specifically, when compared to the baseline, running applications on eventual consistency with

monitors (but without rollback) improved the application throughput by 20%–80%. The benefits are high when the percentage of *PUT* requests is high or when network latency is long. The monitors are efficient in detecting violations. In regional networks (global network, respectively), the monitors were able to detect more than 99.9% of violations in less than 50 milliseconds (3 seconds, respectively) while only introducing a small overhead that was typically less than 4% and in very stressed cases less than 8%. This efficiency allows the monitors themselves to be utilized for the sake of distributed debugging and runtime monitoring. Overall, the detect-rollback improved the computation progress of the applications by 10%–50% when compared to running the applications on sequential consistency. In non-terminating applications, the benefit was 45%–47% while in terminating applications the benefit was 10%–20%. One of the reasons for the reduced benefits in terminating applications is that those applications suffer extensive livelocks during their last phase of computation. We proposed the rollback with an adaptive consistency mechanism which switches from eventual consistency to sequential consistency if livelocks are detected to avoid computation stall.

In the self-stabilization approach, neither the monitors nor the rollback mechanism is needed as cvfs can be handled by the convergence property of self-stabilizing programs. Running a self-stabilizing program on eventual consistency has the advantage of higher throughput and the disadvantage of perturbations caused by cvf. Our experiments with the maximal-matching self-stabilizing program show that the gain is higher than the cost. Specifically, we observed that running the self-stabilizing maximal-matching program on eventual consistency helped the program converge 1.2–1.8 times faster. The benefits scaled when we increased the level of concurrency. Furthermore, when we drop the locking mechanism among the clients and treated mutual exclusion violations as cvfs, the chance of cvf perturbation is higher but the overhead of locking is also eliminated. Indeed, the speedup benefits observed in this case were even higher, as much as 7–12 times.

To compare the detect-rollback and self-stabilization approach, we evaluated their benefits on three self-stabilizing programs: planar graph coloring, general graph coloring, and maximal-

matching. The evaluation results show that the self-stabilization approach provides substantially more benefits than the detect-rollback approach. The stabilization improved the convergence time of the case study programs by 25%–35%. Notably, the aggressive stabilization (that trades additional cvfs for no locking overhead) reduced the convergence time 2–15 times. By contrast, the detect-rollback approach provided limited benefits and potentially causes performance to suffer when compared with sequential consistency. We also considered non-stabilizing programs. The results on stabilizing and non-stabilizing programs for graph coloring show that the stabilizing algorithm is less efficient than the non-stabilizing counterpart on sequential consistency but is the overall winner when used with eventual consistency. The results indicate that there may be a substantial benefit in revising an existing algorithm for the problem at hand to make the algorithm stabilizing and reduce the overall runtime under eventual consistency.

We also considered mechanisms to improve the performance of stabilization. In particular, we used randomization to overcome livelocks in aggressive stabilization and proposed heuristics to reduce the execution time of stabilizing programs in case the convergence pattern exhibits a long tail of slow progress at the end. We also analyzed the effect of several factors such as the input graph structure, the partitioning scheme, and the characteristics of the stabilizing program, on the performance of the detect-rollback and self-stabilization approaches. We observe their effects on the two approaches are different. For example, input graphs with complex structure such as social graphs impose high locking overhead and a high chance of client conflicts. Consequently, the detect-rollback approach performs poorly on social graphs while the aggressive stabilization approach (which removes the locking overhead) performs well. However, on regular graphs with a small average degree, the detect-rollback performs well while the benefits of aggressive stabilization are reduced.

The results presented in this dissertation indicate that there are benefits in employing the optimistic execution approaches. Eventual consistency provides significantly higher performance than sequential consistency. Although eventual consistency introduces cvfs, in many cases these cvfs could be handled efficiently to ensure the correctness of the computation.

APPENDIX

APPENDIX

PUBLICATIONS AND AVAILABILITY OF THE SOURCE CODE AND EXPERIMENTAL RESULTS

Conference Papers

1. D. Nguyen and S. S. Kulkarni. Benefits of Stabilization versus Rollback in Self-Stabilizing Graph-Based Applications on Eventually Consistent Key-Value Stores. *2020 International Symposium on Reliable Distributed Systems (SRDS),* Shanghai, China, 2020, pp. 11-20, doi: 10.1109/SRDS51746.2020.00009.

2. Duong N. Nguyen, Sandeep S. Kulkarni, and Ajoy K. Datta. Benefit of self-stabilizing protocols in eventually consistent key-value stores: a case study. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 148–157, 2019.

3. D. Nguyen, A. Charapko, S. S. Kulkarni, and M. Demirbas. Using weaker consistency models with monitoring and recovery for improving performance of key-value stores. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, pages 67–76, Oct 2018

4. Sorrachai Yingchareonthawornchai, Duong N. Nguyen, Vidhya Tekken Valapil, Sandeep S. Kulkarni, and Murat Demirbas. Precision, recall, and sensitivity of monitoring partially synchronous distributed systems. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 420–435, 2016.

Journal Papers

1. Duong N. Nguyen, Aleksey Charapko, Sandeep S. Kulkarni, and Murat Demirbas. Using weaker consistency models with monitoring and recovery for improving performance of key-

value stores. *J Braz Comput Soc* 25, 10 (2019). https://doi.org/10.1186/s13173-019-0091-9.

2. Sorrachai Yingchareonthawornchai, Duong N. Nguyen, Sandeep S. Kulkarni, Murat Demirbas: Analysis of Bounds on Hybrid Vector Clocks. *IEEE Trans. Parallel Distrib. Syst.* 29(9): 1947-1960 (2018)

Manuscript in Preparation

1. Duong N. Nguyen, Sorrachai Yingchareonthawornchai, Vidhya Tekken Valapil, Sandeep S. Kulkarni, and Murat Demirbas. Precision, recall, and sensitivity of monitoring partially synchronous distributed systems (*Journal manuscript under review*).

Availability of Source Code and Experimental Results

The experimental data and the source code used in chapters 3, 4, and 5 of this dissertation are available at [144], [145], and [146], respectively.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Sukumar Ghosh. *Distributed systems: an algorithmic approach*. CRC press, 2014.

[2] Vijay K Garg and Craig M Chase. Distributed algorithms for detecting conjunctive predicates. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 423–430. IEEE, 1995.

[3] Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[5] Project voldemort. http://www.project-voldemort.com/voldemort/quickstart.html. Accessed: 2020-Nov-20.

[6] LinkedIn Engineering. Venice articles. https://engineering.linkedin.com/blog/topic/venice. Accessed: 2019-4-5.

[7] Amazon dynamodb – a fast and scalable nosql database service designed for internet scale applications. http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html. Accessed: 2019-08-10.

[8] Azure cosmos db – globally distributed database service. https://azure.microsoft.com/en-us/services/cosmos-db/?v=17.45b. Accessed: 2019-08-10.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[11] Shlomi Dolev. *Self-Stabilization.* The MIT Press. The MIT Press, 2000.

[12] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[13] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[14] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, page 7, 2000.

[15] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[16] Hesam Nejati Sharif Aldin, Hossein Deldari, Mohammad Hossein Moattar, and Mostafa Razavi Ghods. Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications. *CoRR*, abs/1902.03305, 2019.

[17] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *International Conference on Networked Systems*, pages 175–189. Springer, 2013.

[18] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[19] Mohammad Roohitavaf and Sandeep S. Kulkarni. DKVF: a framework for rapid prototyping and evaluating distributed key-value stores. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 912–915, 2018.

[20] Azure cosmos db. https://azure.microsoft.com/en-us/services/cosmos-db/. Accessed: 2020-Nov-20.

[21] An introduction to redis data types and abstractions. https://redis.io/topics/data-types-intro. Accessed: 2020-Nov-20.

[22] Couchbase db documentation. https://docs.couchbase.com/java-sdk/current/howtos/kv-operations.html. Accessed: 2020-Nov-20.

[23] Arangodb: The many faces of a native multi-model database. https://www.arangodb.com/why-arangodb/multi-model/. Accessed: 2020-Nov-20.

[24] Apache hbase reference guide. https://hbase.apache.org/book.html. Accessed: 2020-Nov-20.

[25] Michel Raynal. *Distributed algorithms for message-passing systems*, volume 500. Springer, 2013.

[26] Aleksey Charapko, Ailidani Ailijiang, Murat Demirbas, and Sandeep Kulkarni. Retrospective lightweight distributed snapshots using loosely synchronized clocks. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2061–2066. IEEE, 2017.

[27] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[28] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2):109–117, 1992.

[29] Karine Altisen, Ajoy K. Datta, Stéphane Devismes, Anaïs Durand, and Lawrence L. Larmore. Leader election in asymmetric labeled unidirectional rings. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 182–191. IEEE Computer Society, 2017.

[30] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011.

[31] Michiko Inoue, Fukuhito Ooshita, and Sébastien Tixeuil. Brief announcement: Efficient self-stabilizing 1-maximal matching algorithm for arbitrary networks. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 411–413. ACM, 2017.

[32] Ajoy Kumar Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Maximum matching for anonymous trees with constant space per process. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[33] Hisaki Kobayashi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Brief announcement: A self-stabilizing algorithm for the minimal generalized dominating set problem. In Paul G. Spirakis and Philippas Tsigas, editors, *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, volume 10616 of *Lecture Notes in Computer Science*, pages 378–383. Springer, 2017.

[34] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[35] Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem, 2002.

[36] Stephen T. Hedetniemi, David P. Jacobs, and K. E. Kennedy. Linear-time self-stabilizing algorithms for disjoint independent sets, 2012.

[37] Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive self-stabilizing k-clustering. *Theor. Comput. Sci.*, 626:110–133, 2016.

[38] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. A self-stabilizing $O(k)$-time k-clustering algorithm. *Comput. J.*, 53(3):342–350, 2010.

[39] Eddy Caron, Ajoy Kumar Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm using an arbitrary metric. In Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, volume 5704 of *Lecture Notes in Computer Science*, pages 602–614. Springer, 2009.

[40] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336 – 1345, 2009. Structural Information and Communication Complexity (SIROCCO 2007).

[41] D. Nguyen, A. Charapko, S. S. Kulkarni, and M. Demirbas. Using weaker consistency models with monitoring and recovery for improving performance of key-value stores. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, pages 67–76, Oct 2018. Extended version is available at http://arxiv.org/abs/1909.01980.

[42] Duong N. Nguyen, Aleksey Charapko, Sandeep S. Kulkarni, and Murat Demirbas. Using weaker consistency models with monitoring and recovery for improving performance of key-value stores. *J. Braz. Comp. Soc.*, 25(1):10:1–10:25, 2019.

[43] Duong N. Nguyen, Sandeep S. Kulkarni, and Ajoy K. Datta. Benefit of self-stabilizing protocols in eventually consistent key-value stores: a case study. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 148–157, 2019.

[44] D. Nguyen and S. S. Kulkarni. Benefits of stabilization versus rollback in self-stabilizing graph-based applications on eventually consistent key-value stores. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20, 2020.

[45] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[46] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. In Kerry Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, pages 56–66, 1988.

[47] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[48] Murat Demirbas and Sandeep Kulkarni. Beyond truetime: Using augmentedtime for improving google spanner. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2013.

[49] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *International Workshop on Distributed Algorithms*, pages 254–272. Springer, 1991.

[50] Scott D Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Computing*, 13(2):85–98, 2000.

[51] Duong Nguyen and Sandeep S. Kulkarni. Technical report: Benefits of stabilization versus rollback in self-stabilizing graph-based applications on eventually consistent key-value stores. *CoRR*, 2020.

[52] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

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

[54] Vijay K Garg. *Principles of distributed systems*, volume 3144. Springer Science & Business Media, 2012.

[55] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[56] CJ Bovy, HT Mertodimedjo, G Hooghiemstra, H Uijterwaal, and Piet Van Mieghem. Analysis of end-to-end delay measurements in internet. In *Proc. of the Passive and Active Measurement Workshop-PAM*, volume 2002. sn, 2002.

[57] NIST/SEMATECH e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/eda/section3/eda366b.htm, 2013. Accessed: 2019-02-10.

[58] Overview of networkx. https://networkx.github.io/documentation/stable/. Accessed: 2020-May-20.

[59] Jerzy Brzezinski and Dariusz Wawrzyniak. Consistency requirements of peterson's algorithm for mutual exclusion of N processes in a distributed shared memory system. In *Proceedings of the th International Conference on Parallel Processing and Applied Mathematics-Revised Papers*, PPAM '01, pages 202–209, London, UK, UK, 2002. Springer-Verlag.

[60] Bård Fjukstad, John Markus Bjørndalen, and Otto Anshus. Embarrassingly distributed computing for symbiotic weather forecasts. *Procedia Computer Science*, 18:1217–1225, 2013.

[61] Ravi Prakash, Niranjan G Shivaratri, and Mukesh Singhal. Distributed dynamic channel allocation for mobile computing. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 47–56. ACM, 1995.

[62] Yurai Núnez-Rodrıguez, Henry Xiao, Kamrul Islam, and Waleed Alsalih. A distributed algorithm for computing voronoi diagram in the unit disk graph model. In *Proc. 20th Canadian Conference in Computational Geometry (CCCG'08)*, pages 199–202, 2008.

[63] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[64] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986. ACM, 2010.

[65] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett.*, 16(5):217–219, 1983.

[66] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Active stabilization. In *SSS*, pages 77–91, 2011.

[67] Sven Köhler and Volker Turau. Fault-containing self-stabilization in asynchronous systems with constant fault-gap. *Distributed Computing*, 25(3):207–224, Jun 2012.

[68] Volker Turau. Computing fault-containment times of self-stabilizing algorithms using lumped markov chains. *Algorithms*, 11(5), 2018.

[69] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Fault-containment in weakly-stabilizing systems. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 209–223, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[70] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 189–203, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[71] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 45–54, New York, NY, USA, 1996. ACM.

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

[73] `https://en.wikipedia.org/wiki/Zero_one_infinity_rule`.

[74] Duong Nguyen, Aleksey Charapko, Sandeep Kulkarni, and Murat Demirbas. Technical report: Optimistic execution in key-value store. *CoRR*, 2018.

[75] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), October 2015.

[76] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[77] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center,*

Yorktown Heights, New York, USA, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[78] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489–506, 1979.

[79] Nicolas Barnier and Pascal Brisset. Graph coloring for air traffic flow management. *Annals of operations research*, 130(1):163–178, 2004.

[80] Gregory J Chaitin. Register allocation & spilling via graph coloring. *ACM Sigplan Notices*, 17(6):98–101, 1982.

[81] Andrea Gigli, Fabrizio Lillo, and Daniele Regoli. Recommender systems for banking and financial services. In *RecSys 2017 Poster Proceedings, August 27-31, Como, Italy*, 2017.

[82] Ryan A. Rossi and Nesreen K. Ahmed. Coloring large complex networks. *Social Netw. Analys. Mining*, 4(1):228, 2014.

[83] Wei Liao, Kun Deng, and Shengyuan Wang. Community detection based on graph coloring. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2114–2118. IEEE, 2019.

[84] Pierre Hansen and Michel Delattre. Complete-link cluster analysis by graph coloring. *Journal of the American Statistical Association*, 73(362):397–403, 1978.

[85] Omayya Murad, Azzam Sleit, and Ahmad Sharaiah. Improving friends matching in social networks using graph coloring. *International Journal*, 15(8), 2016.

[86] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

[87] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. Diversified top-k clique search. *The VLDB Journal*, 25(2):171–196, 2016.

[88] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. *Nvidia White Paper*, 2015.

[89] Nikos Armenatzoglou, Huy Pham, Vasilis Ntranos, Dimitris Papadias, and Cyrus Shahabi. Real-time multi-criteria social graph partitioning: A game theoretic approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1617–1628, 2015.

[90] Christoph Lenzen. *Synchronization and symmetry breaking in distributed systems*. PhD thesis, ETH Zurich, 2011.

[91] Damien Imbs, Sergio Rajsbaum, and Michel Raynal. The universe of symmetry breaking tasks. In *International Colloquium on Structural Information and Communication Complexity*, pages 66–77. Springer, 2011.

[92] Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloration and arbitrary graphs. In *Procedings of the 4th International Conference on Principles of Distributed Systems, OPODIS 2000, Paris, France, December 20-22, 2000*, pages 55–70, 2000.

[93] Sukumar Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.

[94] ST Hedetniemi and S Mitchell. Edge domination in trees. In *Proceeding of the 8th Southeastern Conference on Combinatorics, Graph Theory and Computing, Louisiana State Univ., Baton Rouge, La.*, volume 19, pages 489–509, 1977.

[95] Michel Balinski and Tayfun Sönmez. A tale of two mechanisms: student placement. *Journal of Economic theory*, 84(1):73–94, 1999.

[96] Donald Ervin Knuth. *Stable marriage and its relation to other combinatorial problems: An introduction to the mathematical analysis of algorithms*, volume 10. American Mathematical Soc., 1997.

[97] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990.

[98] Éric Fusy. Uniform random sampling of planar graphs in linear time. *Random Structures and Algorithms*, 35(4):464–522, 2009.

[99] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[100] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013.

[101] Vito Latora, Vincenzo Nicosia, and Giovanni Russo. *Complex networks: principles, methods and applications*. Cambridge University Press, 2017.

[102] Sorrachai Yingchareonthawornchai, Duong N. Nguyen, Vidhya Tekken Valapil, Sandeep S. Kulkarni, and Murat Demirbas. Precision, recall, and sensitivity of monitoring partially synchronous distributed systems. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 420–435, 2016.

[103] Mohamed G. Gouda and Ted Herman. Adaptive programming. *IEEE Trans. Software Eng.*, 17(9):911–921, 1991.

[104] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[105] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[106] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[107] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[108] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[109] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[110] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[111] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[112] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[113] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[114] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[115] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[116] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[117] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[118] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[119] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[120] Vijay K Garg and Brian Waldecker. Detection of unstable predicates in distributed programs. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 253–264. Springer, 1992.

[121] Vijay K Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, 1996.

[122] Vijay K Garg, Craig M Chase, J Roger Mitchell, and Richard Kilgore. Conjunctive predicate detection. In *Proceedings Hawaii International Conference on System Sciences HICSS95 (January 1995), IEEE Computer Society*. Citeseer, 1995.

[123] Scott D Stoller, Leena Unnikrishnan, and Yanhong A Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *International Conference on Computer Aided Verification*, pages 264–279. Springer, 2000.

[124] Neeraj Mittal and Vijay K Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

[125] Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 101–110. IEEE, 2013.

[126] Xinli Wang, Jean Mayo, Guy Hembroff, and Chunming Gao. Detection of conjunctive stable predicates in dynamic systems. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 828–835. IEEE, 2009.

[127] Xinli Wang, Jean Mayo, and Guy C Hembroff. Detection of a weak conjunction of unstable predicates in dynamic systems. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 338–346. IEEE, 2010.

[128] Vidhya Tekken Valapil and Sandeep S. Kulkarni. Biased clocks: A novel approach to improve the ability to perform predicate detection with O(1) clocks. In *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Ma'ale HaHamisha, Israel, June 18-21, 2018, Revised Selected Papers*, pages 345–360, 2018.

[129] Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep S. Kulkarni, Eric Torng, and Murat Demirbas. Monitoring partially synchronous distributed systems using SMT solvers. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, pages 277–293, 2017.

[130] Weijia Song, Theo Gkountouvas, Ken Birman, Qi Chen, and Zhen Xiao. The freeze-frame file system. In *SoCC*, pages 307–320, 2016.

[131] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[132] Fernando Chirigati, Jérôme Siméon, Martin Hirzel, and Juliana Freire. Virtual lightweight snapshots for consistent analytics in nosql stores. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1310–1321. IEEE, 2016.

[133] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[134] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, 2014.

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

[136] Mohamed G. Gouda. The theory of weak stabilization. In *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, pages 114–123, 2001.

[137] Ted Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[138] Mohammad Roohitavaf and Sandeep S. Kulkarni. Collaborative stabilization. In *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*, pages 259–268. IEEE Computer Society, 2016.

[139] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 45–54, New York, NY, USA, 1996. ACM.

[140] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Fault-containment in weakly-stabilizing systems. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 209–223, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[141] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In Toshimitsu Masuzawa and Sébastien Tixeuil, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 189–203, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[142] Sven Köhler and Volker Turau. Fault-containing self-stabilization in asynchronous systems with constant fault-gap. *Distributed Computing*, 25(3):207–224, Jun 2012.

[143] Volker Turau. Computing fault-containment times of self-stabilizing algorithms using lumped markov chains. *Algorithms*, 11(5), 2018.

[144] Duong Nguyen. Supplementary dataset and source code for the paper "Using Weaker Consistency Models with Monitoring and Recovery for Improving Performance of Key-Value Stores". https://doi.org/10.5281/zenodo.3338381, July 2019.

[145] Duong Nguyen. Supplementary materials for the paper "Benefit of Self-Stabilizing Protocols in Eventually Consistent Key-Value Stores: A Case Study". https://doi.org/10.5281/zenodo.4449643, January 2021.

[146] Duong Nguyen. Supplementary materials for the paper "Benefits of Stabilization versus Rollback in Self-Stabilizing Graph-Based Applications on Eventually Consistent Key-Value Stores". https://doi.org/10.5281/zenodo.3606271, January 2020.