RESILIENT DISTRIBUTED ALGORITHMS FOR SOLVING LINEAR ALGEBRAIC EQUATIONS IN FAULTY NETWORKS

by

Oğuzhan Çiftçi

B.S., Electrical and Electronics Engineering, Marmara University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering

Boğaziçi University

2022

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, Prof. Mehmet Akar, for his guidance, patience, encouragement, and useful suggestions throughout this thesis study.

I also would like to express my special thanks to Assoc. Prof. Onur Cihan for his motivation and advice during my study and also for being a part of my thesis jury.

This thesis is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Project 117E204.

Most importantly, I am grateful for the unconditional, sincere, and loving support of my wife, Merve Çiftçi, and my parent, Nadide Çiftçi, Nurettin Çiftçi, Hacer Çelik, Hüseyin Çelik, not only during my study but also throughout my life.

ABSTRACT

RESILIENT DISTRIBUTED ALGORITHMS FOR SOLVING LINEAR ALGEBRAIC EQUATIONS IN FAULTY NETWORKS

Various methods have been developed to solve linear algebraic equations distributively over multi-agent networks. Most studies consider that all agents are trustworthy and utilize all the received data from their neighbors throughout the process. Nevertheless, cooperation between non-faulty agents is disrupted if faulty agents intrude into the network. This thesis aims to develop algorithms to detect all faulty agents in the network without prior knowledge of the number of faulty agents. We study four fault models: *random-state*, *fixed-state*, *single-faced*, and *double-faced* and propose fault detection procedures according to the characteristics of these fault models. First, we introduce a method in which each agent can determine its neighbors' system of equations if it receives sufficient solution estimations from neighboring agents. By utilizing this method, we propose a synchronous discrete-time distributed detection algorithm for the perfectly synchronized agents in terms of their event times. On the other hand, the event time sequences of different agents are not always assumed to be synchronized. Therefore, we also propose an asynchronous discrete-time distributed fault detection algorithm to analyze the effect of the asynchronous event times of agents. Also, we discuss the applicability of our detection algorithm in continuous-time systems. Moreover, complexity analyses for the proposed algorithms are carried out. Theoretical results are also illustrated by numerical examples.

ÖZET

DOĞRUSAL CEBİR DENKLEMLERİNİN HATALI AĞLARDA ÇÖZÜMÜ İÇİN DİRENÇLİ DAĞITIK ALGORİTMALAR

Doğrusal cebir denklemlerini çok etmenli ağlar üzerinden dağıtık olarak çözmek için çeşitli yöntemler geliştirilmiştir. Çoğu çalışma, tüm etmenlerin güvenilir olduğunu ve süreç boyunca komşulardan alınan tüm verilerin kullanıldığını kabul eder. Bununla birlikte, hatalı etmenlerin ağa sızmaları durumunda, hatalı olmayan etmenler arasındaki işbirliği de bozulmaktadır. Bu tezde, hatalı etmenlerin sayısı hakkında önceden bilgi sahibi olmadan, ağdaki tüm hatalı etmenlerin tespit edilmesi için algoritmalar geliştirilmesi amaçlanmıştır. Bunun için, dört hata modeli sunulmuştur: *rastgele-durum*, *sabit-durum*, *tek-yüzlü* ve *çift-yüzlü*. Bu hata modellerinin özelliklerine göre hata tespit prosedürleri önerilmiştir. Öncelikle, her bir etmenin komşusundan yeterli çözüm verisini almasıyla, komşusunun denklem sistemini belirleyebildiği bir yöntem tanıtılmıştır. Bu yöntemden faydalanılarak, olay zamanları açısından mükemmel bir şekilde senkronize edilmiş etmenler için senkronize ayrık-zamanlı dağıtık bir hata tespit algoritması önerilmiştir. Öte yandan, farklı etmenlerin olay zaman dizilerinin her zaman senkronize olduğu varsayılamaz. Bu nedenle, etmenlerin eşzamansız olay zamanlarının etkisini analiz etmek için eşzamansız ayrık-zamanlı dağıtık bir hata tespit algoritması da önerilmiştir. Ayrıca, hata tespit algoritmamızın sürekli zamanlı sistemlerde uygulanabilirliği de tartışılmıştır. Önerilen algoritmalar için karmaşıklık analizleri yapılmıştır. Teorik sonuçlar aynı zamanda sayısal örneklerle doğrulanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

A	The system matrix
a_{ij}	The jth coefficient of ith equation in set of equations
b	The system constant vector
C_j	The augmented matrix with the solution vectors of agent j
D	The diagonal matrix
$\mathcal{E}(t)$	The Time-varying edge set between the nodes
F	The total number of faulty agents in the network
$\mathcal{G}$	The fixed graph
$\mathcal{G}(t)$	The time-varying graph
$\boldsymbol{h}_i$	The update on evolution of the solution estimates of agent i
k	The discrete-time index
L	The lower triangular matrix
M_j	The matrix that its columns span the equation set of agent j
m	The total number of autonomous agents in the network
$\mathbb{N}$	The set of natural values
N	The off-diagonal matrix
$\mathcal{N}_i$	The neighbor set of node i
$\mathcal{N}_i^+$	The in-neighbor set of node i
$\mathcal{N}_i^-$	The out-neighbor set of node i
n	The number of parameters in the linear equation set
$\mathcal{O}$	The Big-O notation
P	The projection matrix
P_{A_i}	The projection matrix of equation set of agent i
$\mathbb{R}$	The set of real values
r	The number of linearly independent equations
r_i	The number of linearly independent equations of agent i
t_{ik}	The kth event time of agent i
U	The strictly upper triangular matrix

$\mathcal{V}$	The set of nodes in the network
$\mathcal{V}_n$	The set of normal nodes in the network
$\mathcal{V}_f$	The set of faulty nodes in the network
v_i	The label of node i
$\boldsymbol{v}_j$	The average of the states of agent j's neighbors
$\boldsymbol{x}_i$	The solution vector of agent i
x_i	The ith parameter of solution vector
$\boldsymbol{x}_i^k$	kth solution estimate of agent i in discrete-time setting
$\boldsymbol{x}_i^j(t_k)$	The solution estimate of agent i sends to agent j at kth iteration
$\boldsymbol{x}_i^j(t_{ik})$	The solution estimate of agent i sends to agent j at kth event time
$\mathbb{Z}$	The set of integer values
δ_{ij}	The time delay between the agent i and agent j
δ_{max}	The maximum time delay in the network
ϵ_c	A small positive algorithm specific precision value
μ_i	The number of in-neighbors of agent i

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
RCC	Resilient Convex Combination
SVD	Singular Value Decompositon
W-MSR	Weighted Mean-Select-Reduced

1. INTRODUCTION

Although almost all systems in nature (reproduction of bacteria habitats, flight dynamics of UAVs, etc.) exhibit non-linear behavior, the systems that involve numerical computations can be modeled by using a set of linear equations of the form

$$A\boldsymbol{x} = b \tag{1.1}$$

where $A \in \mathbb{R}^{m \times n}$ denotes the system matrix, $\boldsymbol{x} \in \mathbb{R}^n$ denotes the solution vector, and $b \in \mathbb{R}^m$ denotes the system's constants.

Many different approaches and algorithms have been developed to solve such equations for decades [1, 2]. However, most early suggested algorithms consider that there is only one centralized solver to compute a solution to the given system of linear equations [3–6].

The centralized methods can be considered practical and cost-efficient for small-scale systems. However, having a single processor to solve a large-scale system of linear equations by using centralized algorithms may lead to catastrophic issues. First, it may be computationally expensive to apply centralized algorithms for applications with many unknowns since all the computations to solve $A\boldsymbol{x} = b$ occur in a single solver [7]. Secondly, a system with a single solver would be vulnerable to cyber-attacks, and hostile parties can easily interfere in the system to reach private information. Thirdly, conventional centralized methods may not directly apply to over-determined systems where a feasible solution cannot be achieved [7–9]. Last but not least, the set of linear equations may not even be solvable by using centralized algorithms since the constraints on the forms of the system matrix A in (1.1) may be pretty strict in some centralized algorithms [1, 2].

Meanwhile, a considerable amount of effort has been given to developing distributed algorithms in the parallel processing community over the past decades. The main motives for using distributed algorithms rather than centralized methods are to

achieve both efficiencies in computational workload for the processors and improve information security by decomposing a large system of linear equations into smaller ones [7, 8, 10].

On the other hand, distributed algorithms seem more prominent in meeting today's needs than centralized algorithms. For instance, a robot swarm can be characterized by a network of multiple mobile and autonomous robots moving around an area to execute collaborative tasks. We assume only wireless communication links are assigned between the robots so that they can transmit information through these wireless links [11]. Also, different energy resources harvested from plants (nuclear plants, solar panels, wind turbines, etc.), can be optimally coordinated using distributed algorithms [12–15]. Thus, we can utilize distributed algorithms to achieve common goals that are hard to be attained by an individual agent using a centralized algorithm. Many other practical applications are reviewed in [16–20].

In distributed algorithms, we think of $m > 1$ autonomous agents in the network, and each agent can compute an initial estimate solution for its own set of equations and exchange information with its *neighboring* agents at each event time. Neighboring relationships among the agents can be shown by a graph representation with m nodes, and a set of edges that illustrates the communication links between the nodes. The graph representation can be a directed graph on which the direction of edges indicates the direction of the information flow. Furthermore, most studies assume that each agent owns only a subset of the overall equation set [21–26].

Nevertheless, most earliest suggested distributed algorithms to solve linear algebraic equations in the literature presume that all the agents are trustworthy. Therefore, agents undoubtedly utilize the states of their neighboring agents. However, if some of the agents send erroneous information to their neighboring agents, faulty agents may drag the normal agents to a solution that is an infeasible solution to $A\boldsymbol{x} = b$. In the worst case, the faulty agents may even lead the non-faulty agents to diverge to infinity in total error. Therefore, an algorithm for solving a large system of linear equations

in a distributed manner should be somehow resilient to faulty agents [27–30]. In this way, normal agents can identify and remove the faulty ones from their neighbor list. Thus, they can aggregate on the same feasible solution to $A\boldsymbol{x} = b$.

As described above, each agent broadcasts information to its neighboring agents at each event time. However, it is not apparent whether the provided event time for the agents is in perfect synchrony. Therefore, the distributed algorithms can be characterized in line with the timing of events of the agents, namely synchronous or asynchronous distributed algorithms. Agents can use the synchronous distributed algorithms if all the agents perform their computation and communication in a fully synchronized way. In most of the synchronous distributed algorithms, transmission delays are ignored [7, 23]. On the other hand, we observe that all agents spend some time on computation and transmission in real applications. Therefore, in this study, we also consider the asynchronous models in which each agent has its own event time to determine and send its solution estimate to its neighbors.

1.1. Motivations of the Thesis

In this thesis, we concentrate on the resilient distributed algorithms to solve linear algebraic equations of the form (1.1) in faulty networks. It is crucial to eliminate the effects of all faulty agents in order to achieve consensus on a feasible solution to $A\boldsymbol{x} = b$.

Most existing studies on resilient distributed consensus only consider reaching a common value as we have for voting and rendezvous problems. Nevertheless, achieving consensus on the same state will not suffice to have a feasible solution for $A\boldsymbol{x} = b$, which is the problem of interest here. Therefore, we study to develop a resilient distributed algorithm that ensures all non-faulty agents aggregate on the same feasible solution to $A\boldsymbol{x} = b$, after suppressing the effects of faulty agents. Moreover, we discuss fault detection in continuous-time networks as well.

Most of the studies in the literature assume that the equation systems of neigh-

boring agents are private due to security concerns [7, 22, 23]. On the other hand, we propose an approach to reveal an equivalent equation system of neighboring agents by using received linearly independent solution estimates.

On the other hand, we know the synchronization among the agents' event times affects the distributed solution estimation of the linear equation systems of the form (1.1) [24]. Thus, we propose resilient distributed algorithms to study the effects of asynchronization on the agents' event times.

1.2. Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2: We first provide some fundamental concepts about graph theory. Then we review two pioneering centralized methods to solve linear algebraic equations. Afterward, we continue to review some existing studies on distributed algorithms for solving linear algebraic equations and distributed fault tolerant consensus algorithms. Finally, we provide basics about the complexity analysis of the algorithms.

Chapter 3: We introduce four fault models in multi-agent systems: *random-state*, *fixed-state*, *single-faced*, and *double-faced*. Note that the first three models can be described as Symmetric faults while the last is an Asymmetric (Byzantine) fault. Then we define fault detection procedures that are directly applicable in the discrete-time distributed algorithms. Furthermore, we discuss the extension of these fault detection procedures to continuous-time distributed algorithms.

Chapter 4: We propose a synchronous fault-resilient discrete-time distributed algorithm for solving linear algebraic equations. Then we introduce pseudocode for the suggested algorithm and analyze the algorithm in terms of the time and space complexities. Lastly, we present a numerical example to illustrate our findings.

Chapter 5: We suggest an asynchronous fault-resilient distributed algorithm in a discrete-time setting. We provide pseudocode for the studied algorithm. Furthermore, we discuss the time and space complexity analyses of the algorithm. Finally, some simulation results regarding the proposed algorithm are presented.

Chapter 6: We present concluding remarks about the topics discussed in this study.

2. MATHEMATICAL PRELIMINARIES AND LITERATURE REVIEW

In this chapter, we introduce some mathematical preliminaries about graph representation. Then we discuss the critical aspects of some pioneering studies on solving linear algebraic equations in centralized systems and distributed systems. Moreover, we will comment on other studies that consider the resiliency of the distributed algorithms in the presence of faulty attacks in the network. Lastly, we comment on the complexity analysis methods.

2.1. Fundamentals of Graph Theory

Communication channels between the agents in a multi-agent network can be expressed mathematically by using graph theory. In this thesis, we consider the terms node and agent in the same sense. We consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, 2, \ldots, m\}$ is the set of nodes and $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the set of directed edges between the nodes. We assume node i and node j are adjacent to each other if a communication link exists between them, i.e., $(i, j) \in \mathcal{E}$. These nodes are also called *neighbors*.

Furthermore, we define the set of nodes that sends information to node i as in-neighbors of node i, i.e., $\mathcal{N}_i^+ = \{j \in \mathcal{V} : (j, i) \in \mathcal{E}\}$. Likewise, the set of nodes that receives information from node i is defined as out-neighbors of node i, i.e., $\mathcal{N}_i^- = \{j \in \mathcal{V} : (i, j) \in \mathcal{E}\}$. We also consider that each node $i \in \mathcal{V}$ has a self-link, i.e., $i \in \mathcal{N}_i^+$.

It is important to note that the individual nodes do not necessarily share their information with all agents. Therefore, the direction of the information flow has a significant role in graph theory. Node i only receives information from the agents in its neighboring set $\mathcal{N}_i^+$.

The order of the nodes reveals the direction of the information flow. For instance, $(i, j) \in \mathcal{E}$ illustrates that node i sends its data to node j. If the information flow is bidirectional, we consider the graph as *undirected*, such that

$$i, j \in \mathcal{V} : (j, i) \in \mathcal{E} \Leftrightarrow (i, j) \in \mathcal{E}, \ \forall \ i, j \in \{1, 2, \ldots, m\}. \tag{2.1}$$

Conversely, a directed edge between any two nodes in $\mathcal{G}$ indicates that the information flow is unidirectional.

As mentioned above, connectivity among the nodes is vital in networked systems. Therefore, the following definitions are provided to help understand some essential network topologies [10].

Definition 2.1. (Time-varying and fixed graphs) *A graph $\mathcal{G}(t) = (\mathcal{V}, \mathcal{E}(t))$ is defined as the time-varying graph if the edge set $\mathcal{E}(t)$ is altered in time while the set of vertices $\mathcal{V}$ remains the same. On the other hand, a digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is called a fixed graph if there is no change in the edge set or vertex set over time.*

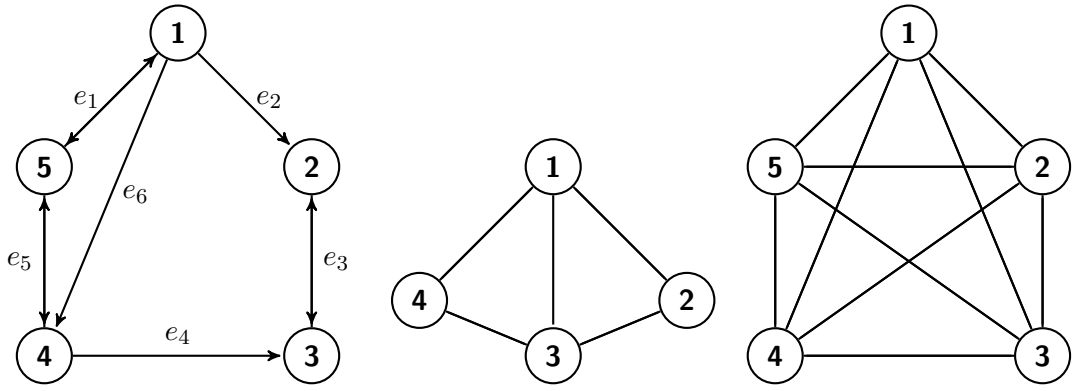
Definition 2.2. (Strongly connected graph) *A graph $\mathcal{G}$ is a strongly connected graph if it is a directed graph and a directed path is formed between each pair of distinct nodes in $\mathcal{V}$.*

Definition 2.3. (Connected graph) *A graph $\mathcal{G}$ is described as a connected graph if it is an undirected graph and a path is formed between each pair of nodes in $\mathcal{V}$.*

Definition 2.4. (Complete graph) *A graph $\mathcal{G}$ is defined as a complete or fully connected graph if it is an undirected graph, and an edge is formed between each pair of distinct nodes. In graph $\mathcal{G}$, there should be a total of $m(m-1)/2$ undirected edges between m nodes.*

Example 2.1. *Consider the graph representations $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ in Figure 2.1.*

$\mathcal{G}_1$ is a directed graph consisting of five nodes and nine directed edges. The arrows on the tip of the edges identify the information flow direction between the nodes. As can be seen from the figure, the edges e_1, e_3 and e_5 can carry information bilaterally while the others can carry only in one way. For instance, node 1 shares its information



(a) $\mathcal{G}_1$ (b) $\mathcal{G}_2$ (c) $\mathcal{G}_3$

Figure 2.1. 3 different graph topologies

with node 4 but not vice versa since the arrow on tip of e_6 *only points to node 4. Furthermore, graph* $\mathcal{G}_1$ *is not strongly connected since there is no directed path from node 2 to node 4.*

$\mathcal{G}_2$ *is an undirected graph with four nodes and five undirected edges. We note that the communication links between the nodes are bidirectional since there is no arrow on the tip of the edges. For instance, node 2 and node 3 share their information bilaterally. Since all the nodes in* $\mathcal{G}_2$ *are connected with undirected edges,* $\mathcal{G}_2$ *is called a* connected *graph.*

$\mathcal{G}_3$ *is an undirected graph that consists of five nodes and ten undirected edges. The significance of the given graph is that all the distinct nodes are connected bilaterally. Thus* $\mathcal{G}_3$ *is a* complete *graph.*

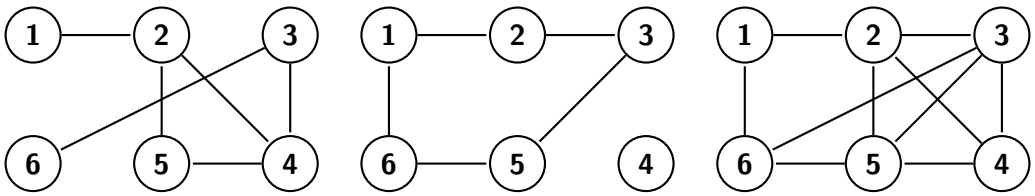
Since the connectivity relations among the nodes can be more cumbersome in real-world applications, the above graph definitions might not be sufficient to analyze the networked systems. Therefore, we shall introduce the following graph-theoretical properties.

Definition 2.5. (Union of graphs) *The union of two graphs* $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ *and* $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ *can be represented as follows:*

$$\mathcal{G}_u = (\mathcal{V}_u, \mathcal{E}_u) = \mathcal{G}_1 \circ \mathcal{G}_2 \tag{2.2}$$

where $\mathcal{V}_u = \mathcal{V}_1 \cup \mathcal{V}_2$ *and* $\mathcal{E}_u = \mathcal{E}_1 \cup \mathcal{E}_2$.

Example 2.2. *Consider the given three graph topologies in Figure 2.2. The two on the left are the individual graph topologies* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *with the same node set* $\mathcal{V}$. *The union of these two graphs* $\mathcal{G}_1 \circ \mathcal{G}_2$ *is illustrated in Figure 2.2(c).*



(a) $\mathcal{G}_1$ (b) $\mathcal{G}_2$ (c) $\mathcal{G}_3 = \mathcal{G}_1 \circ \mathcal{G}_2$

Figure 2.2. The union of graphs.

Definition 2.6. *[22]* (Jointly strongly connected graphs) *Sequence of a finite number of graphs with the same node set* $\mathcal{V}$ *forms a jointly strongly connected graph if the union of the given sequence is a strongly connected graph.*

Definition 2.7. *[22]* (Repeatedly jointly strongly connected graphs) *An infinite sequence of directed graphs* $\mathcal{G}_1,\ \mathcal{G}_2, \ldots$ *with the same node set is called repeatedly jointly strongly connected if there exist finite positive integers* l *and* τ *such that for any integer* $k > 1$, *the finite sequence* $\mathcal{G}_{kl+\tau-1} \circ \mathcal{G}_{kl+\tau-2} \circ \ldots \circ \mathcal{G}_{(k-1)l+\tau}$ *is jointly strongly connected.*

2.2. Centralized Methods

In this section, two of the most recognized centralized methods, the Jacobi method and the Gauss-Seidel method, are briefly introduced [1]. Most techniques in the literature are based on partitioning the square system matrix A in (1.1) with non-zero diagonal elements to devise consistent centralized methods.

2.2.1. Jacobi Method

The Jacobi method was used by solving the ith equation in (1.1), to obtain $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$ in an iterative manner [1]. First of all, we have the linear system of equations as follows

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n. \end{aligned} \tag{2.3}$$

By manipulating the above linear equations individually, each of the unknowns can be expressed as follows:

$$\begin{aligned} x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)}), \\ x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \ldots - a_{2n}x_n^{(k)}), \\ &\vdots \\ x_n^{(k+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \ldots - a_{nn-1}x_{n-1}^{(k)}) \end{aligned} \tag{2.4}$$

where $x_i^{(k)}$ is the solution of ith equation at kth iteration, $k \geq 0$, and the initial state x_i^0 is an arbitrarily chosen initial solution of the ith equation.

Hence, $x_i^{(k+1)}$ can be iteratively computed by the equation of

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ i \neq j}}^{n} a_{ij} x_j^{(k)} \right], \ k \geq 0 \tag{2.5}$$

with an arbitrarily chosen initial solution x_i^0.

Remark 2.1. *(i) This method can be utilized if and only if* $a_{ii} \neq 0, \forall i \in \{1, 2, \ldots, n\}$.

(ii) Let D *be the diagonal matrix of the diagonal entries of matrix* A*, and* N *be formed by the off-diagonal elements of matrix* A*. We can rewrite* (2.5) *in a general form as follows [1]:*

$$\begin{aligned} A\boldsymbol{x} &= b \\ (D+N)\boldsymbol{x} &= b \\ D\boldsymbol{x} &= -N\boldsymbol{x} + b \\ \boldsymbol{x} &= D^{-1}(-N\boldsymbol{x} + b) \\ \boldsymbol{x}^{(k+1)} &= -D^{-1}N\boldsymbol{x}^{(k)} + D^{-1}b. \end{aligned} \tag{2.6}$$

2.2.2. Gauss-Seidel Method

The main distinction from the prior Jacobi Method is that the Gauss-Seidel method also uses the $(k+1)$th iteration in the latter equation [1], i.e.

$$\begin{aligned} x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)}), \\ x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \ldots - a_{2n}x_n^{(k)}), \\ x_3^{(k+1)} &= \frac{1}{a_{33}}(b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - \ldots - a_{3n}x_n^{(k)}), \\ &\vdots \\ x_n^{(k+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - \ldots - a_{nn-1}x_{n-1}^{(k+1)}). \end{aligned} \tag{2.7}$$

In the Gauss-Seidel method, $x_i^{(k+1)}$ can be computed iteratively from

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right], \ k \geq 0 \tag{2.8}$$

with an arbitrarily chosen initial solution vector x_i^0.

Remark 2.2. *(i) This method can be applied if and only if* $a_{ii} \neq 0, \forall i \in \{1, 2, \ldots, n\}$.

(ii) Let U *be the strictly upper triangular matrix formed from the matrix* A *and* L *be the lower triangular matrix formed from the matrix* A*, such that* $A = U + L$*. We can rewrite* (2.8) *in a general form as follows [1]:*

$$\begin{aligned} A\boldsymbol{x} &= b \\ (U + L)\boldsymbol{x} &= b \\ L\boldsymbol{x} &= -U\boldsymbol{x} + b \\ \boldsymbol{x} &= L^{-1}(-U\boldsymbol{x} + b) \\ \boldsymbol{x}^{(k+1)} &= -L^{-1}U\boldsymbol{x}^{(k)} + L^{-1}b. \end{aligned} \tag{2.9}$$

2.3. Distributed Algorithms to Solve Linear Algebraic Equations

In this section, we briefly discuss some of the pioneering studies in distributed algorithms for solving the linear algebraic equations of the form (1.1) in multi-agent networks. It is assumed that each agent can perform necessary vector and matrix operations such as matrix inversion and vector multiplication and store the necessary data in its memory. Moreover, each agent sends its data of solution estimate to its neighbors. It should also be noted that each agent i knows one or more rows of the augmented matrix $[A\ b]$ but not the entire equation set.

In order to analyze multi-agent networks, we consider that each individual agent is presented as a node in the graph representation. Moreover, the communication links among the nodes at time step t describe the edge set of the network, i.e. $\mathcal{E}(t)$. Also, we comment on the topology definitions for the reviewed algorithm since the convergence analysis of the algorithms may differ in line with the graph definitions.

Distributed algorithms to solve an equation of the form (1.1) in multi-agent networks can be categorized into two time settings: discrete-time and continuous-time.

2.3.1. Discrete-time Distributed Algorithms

In the discrete-time distributed algorithms, we consider that each agent broadcasts its solution estimate to its neighboring agents at discrete time-steps. Therefore, we may presume that there is no time latency in the data transmission among the agents, and the event times for all agents are perfectly synchronized. However, if there is no common clock for all agents, it is claimed that the agents' event times may differ in [24]. Therefore, discrete-time distributed algorithms must be investigated for both of the synchronization conditions.

2.3.1.1. Synchronous Algorithms. In [21], a distributed algorithm to solve the problem of (1.1) was suggested by Pasqualetti *et al.* for fixed and connected graphs. It is stated that the initial estimate for all agents, $\boldsymbol{x}_i(0)$, should be explicitly chosen. Otherwise, the consensus cannot be assured among the agents. During the process, each agent synchronously publishes three particular data: its current estimates $\boldsymbol{x}_i(t)$, the projection matrices of $ker(A_i)$ and $ker(A_j)$ where $j \in \mathcal{N}_i$, which is very expensive in terms of the communication load. However, the propagation of its private information violates the privacy requirements.

Another synchronous discrete-time distributed algorithm has been proposed by Mou *et al.* [22] to solve (1.1) in a repeatedly jointly strongly connected network. The initial estimates of agent i should refer to its own problem set $A_i\boldsymbol{x}_i(t) = b_i$. Then, each agent sends only the current estimate of itself at each iteration t, namely $\boldsymbol{x}_i(t)$. The synchronous distributed algorithm suggested in [22], has the following form:

$$\boldsymbol{x}_i(t+1) = \boldsymbol{x}_i(t) - P_{ker(A_i)} \sum_{j \in \mathcal{N}_i^+(t)} w_{ij}(t)\Big((\boldsymbol{x}_i(t) - \boldsymbol{x}_j(t)\Big), t \geq 1 \tag{2.10}$$

where $w_{ij}(t)$ is the weighting coefficient which was designed to be $\left|\mathcal{N}_i^+(t)\right|^{-1}$, $\forall\, i \in \mathcal{V}$, $P_{ker(A_i)}$ is the projection matrix onto the kernel of A_i which can be computed as follows:

$$P_{ker(A_i)} = I_n - A_i^T\left(A_iA_i^T\right)^{-1}A_i. \tag{2.11}$$

Algorithm (2.10) originates from the agreement principle articulated in [22]. This principle expresses the necessity for each agent $i \in \mathcal{V}$ to achieve a state that solves its own problem set $[A_i\ b_i]$ at each iteration t, and to aggregate on the same state altogether with the other agents that solve the whole system of equations, $A\boldsymbol{x} = b$ by using a consensus algorithm.

When there is a unique solution for $A\boldsymbol{x} = b$ in a repeatedly jointly strongly connected network, algorithm (2.10) is proved to be convergent. Furthermore, when there are infinitely many solutions for $A\boldsymbol{x} = b$, the convergence of the algorithm (2.10) is still guaranteed under the same assumptions on the network topology.

Some studies [23, 24, 31, 32] in the literature analyze the convergence rate of the algorithm (2.10) in terms of different types of communication topologies. For example, the effect of time-varying topologies on the convergence rate is studied in [31]. Furthermore, the gossip communication effect on the convergence rate is analyzed in [23]. In addition, it is shown in [32] that the linear convergence rate can be achieved by the algorithm (2.10) in a more general communication topology with some extensions on other conditions. On the other hand, the condition on the initialization of the estimates of individual agents was relaxed to be selected arbitrarily in [24].

2.3.1.2. Asynchronous Algorithms. In asynchronous distributed algorithms, each agent may deliver its data in delayed time sequences, which specify the computation and transmission delay of the information exchange among the neighboring agents. These time delays are considered as asynchrony in the agent's event times in [24]. The authors proposed two asynchronous distributed algorithms in [24]. In order to have a consensus on the same feasible solution to the equation of the form (1.1) exponentially fast, both algorithms in [24] require the underlying network topology to be repeatedly jointly strongly connected. Furthermore, each agent knows only its own equation set $A_i\boldsymbol{x}_i(t) = b_i$ and broadcasts only the estimate that solves its equation set at each event

time. The first algorithm is formed as follows:

$$\boldsymbol{x}_i(t_{i(k+1)}) = \boldsymbol{x}_i(t_{ik}) - P_{ker(A_i)} \sum_{j\in\mathcal{N}_i(t_{ik})} \Big(\boldsymbol{x}_i(t_{ik}) - \frac{1}{\mu_i(t_{ik})}\boldsymbol{x}_j\big(t_{ik} - \delta_{ij}(k)\big)\Big), k \geq 1 \quad (2.12)$$

where $\mu_i(t_{ik})$ is the number of neighbors of agent i at event time t_{ik}, and $\delta_{ij}(k)$ is the delay time defined as the sum of the *hold time* and the *transmission time*. Moreover, the algorithm (2.12) requires each agent to have an initial state vector $\boldsymbol{x}_i(t_{i1})$ which is a solution to its own equation set, i.e., $A_i\boldsymbol{x}_i(t_{i1}) = b_i$.

On the other hand, the second algorithm suggested in [24] offers relaxation on the requirement imposed in the algorithm (2.12) for the initialization scheme. In the second algorithm, each agent can initialize its state vector at its first event time $\boldsymbol{x}_i(t_{i1})$ as an arbitrary vector in $\mathbb{R}^n$.

2.3.2. Continuous-time Distributed Algorithms

Most of the proposed continuous-time distributed algorithms for solving linear algebraic equations work similarly as the discrete-time algorithms [7]. They combine the conventional consensus algorithms with data of local equations $A_i\boldsymbol{x} = b_i$. The generic form of the continuous-time distributed algorithms is as follows:

$$\dot{\boldsymbol{x}}_i = -\mathbf{G}_i\big(A_i\boldsymbol{x}_i - b_i\big) - \sum_{j\in\mathcal{N}_i} \big(\boldsymbol{x}_i - \boldsymbol{x}_j\big) \quad (2.13)$$

where $\mathbf{G}_i$ is an algorithm-specific matrix.

A projected consensus algorithm is suggested in [33] for solving $A\boldsymbol{x} = b$ distributively in continuous time setting as follows:

$$\dot{\boldsymbol{x}}_i = -P_{ker(A_i)} \sum_{j\in\mathcal{N}_i} \big(\boldsymbol{x}_i - \boldsymbol{x}_j\big) \quad (2.14)$$

where the consensus term is defined as $\sum_{j\in\mathcal{N}_i(k)} \big(\boldsymbol{x}_i - \boldsymbol{x}_j\big)$ and $P_{ker(A_i)}$ is the projection matrix onto $ker(A_i)$. Each agent initializes their estimates at a solution to its local equation. Moreover, a linear rate convergence in the fixed connected graphs is shown in [33].

On the other hand, an initialization-free algorithm is proposed in [33] as follows:

$$\dot{\boldsymbol{x}}_i = -A_i^T\big(A_iA_i^T\big)^{-1}\big(A_i\boldsymbol{x}_i - b_i\big) - P_{ker(A_i)}\sum_{j\in\mathcal{N}_i}\big(\boldsymbol{x}_i - \boldsymbol{x}_j\big). \tag{2.15}$$

In order to achieve this initialization-free algorithm, information of local equations is added to the projected consensus term in algorithm (2.14). The communication topology for the network is defined as fixed and connected as provided in (2.14). Moreover, continuous-time systems were analyzed in [34] when the topology is assumed to be a piecewise-constant switching directed graph.

2.4. Distributed Fault Tolerant Consensus Algorithms

In this section, we review the algorithms developed to eliminate faulty agents' effect in a network. We first examine the resilient Weighted-Mean-Subsequence-Reduced algorithm proposed in [27]. Secondly, we review the method of achieving a resilient convex combination (RCC) suggested in [35].

2.4.1. Weighted-Mean-Subsequence-Reduced (W-MSR) Algorithm

The fault tolerant W-MSR Algorithm is suggested by Leblanc *et al.* [27]. In the W-MSR Algorithm, each normal agent $i \in \mathcal{V}_n$ receives the states of its neighboring agents at time-step t. The maximum number of faulty agents is limited by F, but initially identities of agents are not known by the normal agent i. In order to eliminate the effect of the faulty agents, normal agent i erases the extreme agents from its neighboring set. In this way, agent i can update its value without having to utilize the faulty state. The algorithm was organized as follows:

(i) At each time step t, each normal agent i receives the states of its neighbors and generates a list which is sorted in decreasing order.

(ii) If there are less than F values, strictly larger than its own state, $\boldsymbol{x}_i(t)$, then normal agent i erases all states that are strictly larger than its own. Otherwise, it erases precisely the largest F states in the sorted list (breaking ties arbitrarily).

Likewise, if there are less than F states strictly smaller than its own state, then agent i erases all states that are strictly smaller than its own. Otherwise, it erases precisely the smallest F states.

(iii) Suppose $\mathcal{R}_i(t)$ indicates the set of agents whose states are erased by normal agent i in step (ii) at time-step t. Each normal agent i performs the consensus update as

$$\boldsymbol{x}_i(t+1) = \sum_{j \in \mathcal{N}_i(t) \setminus \mathcal{R}_i(t)} w_{ij}(t) \boldsymbol{x}_j(t), \tag{2.16}$$

where the weights $w_{ij}(t)$ meet the conditions as follows:

- $w_{ij}(t) = 0$ whenever $i \in \mathcal{V}, j \neq \mathcal{N}_i(t), t \in \mathbb{N}$;
- $w_{ij}(t) \geq \alpha > 0, i \in \mathcal{V}, \forall j \in \mathcal{N}_i(t), t \in \mathbb{N}$;
- $\sum_{j=1}^{n} w_{ij}(t) = 1, \forall i \in \mathcal{V}, t \in \mathbb{N}$.

2.4.2. Resilient Convex Combination (RCC) Method

The Resilient Convex Combination or RCC method is a method introduced by Wang *et al.* [35]. The key idea of this method is to achieve a convex combination of the states of the normal agents in the presence of the faulty agents. The normal agents only know the upper limit to the number of faulty agents in $\mathcal{G}$.

Definition 2.8. *[35]* (Resilient convex combination) *Let $\boldsymbol{x}_{\mathcal{A}} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ denote a set of vectors in $\mathbb{R}^n$, where $\mathcal{A} = \{1, 2, \ldots, m\}$. Suppose each agent knows that at the utmost a number of $\mathcal{H}$ vectors in $\boldsymbol{x}_{\mathcal{A}}$ are faulty with the identity tags of the faulty ones hidden. Then there are at least a number of $p = m - \mathcal{H}$ normal vectors in $\boldsymbol{x}_{\mathcal{A}}$. A vector is a resilient convex combination of vector set of $\boldsymbol{x}_{\mathcal{A}}$, if it is a convex combination of at least p normal vectors in $\boldsymbol{x}_{\mathcal{A}}$.*

Although the RCC of the normal agents is well-defined, it is not obvious enough to achieve a convex combination. Therefore, *Tverberg points* method introduced in [36] may be used to achieve an RCC. However, the Tverberg point approach deemed inefficient since the computational complexity of computing Tverberg points grows exponentially with n. Therefore, it was indicated that an RCC can be achieved through the intersection of convex hulls of the vectors in $\boldsymbol{x}_{\mathcal{A}}$. In order to achieve an RCC,

an algorithm based on an optimization problem is introduced in [35]. Since the proposed quadratic optimization problem is somehow cumbersome to represent here, we recommend reviewing the study in [35] for more details.

2.5. Complexity Analysis

In order to compare different algorithms, we can analyze their complexities in terms of run time and space allocation in the memory of the processor. It should be noted that the exact operation count is not necessary and possible for the algorithms. Instead, we can perform an approximation with the highest degree on the total time and space costs. This concept is known as *Big-O notation*, i.e., $\mathcal{O}(\cdot)$ [10].

Definition 2.9. *[10]* (Time Complexity) *Time complexity analysis can be utilized to determine the amount of time spent on executing an algorithm in terms of the length of the inputs. However, it does not indicate the actual run time of the processor to execute the whole algorithm.*

Definition 2.10. *[10]* (Space Complexity) *Space complexity of an algorithm determines the total amount of allocated memory space in the processor during the execution. Also, the input values should be included in the total cost.*

Example 2.3. *Consider Figure 2.3, which presents an algorithm to find the sum of all elements in an input array A of size $n \times 1$.*

Input: An array A of size $n \times 1$

Output: *sum*

1: $sum \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
3: $\quad sum \leftarrow sum + A[i]$
4: **end for**

Figure 2.3. Algorithm to sum of all elements in an array.

We compute the total cost to run the given algorithm by analyzing each statement in the code line by line. We calculate the time cost of assigning a single element to a

variable is 1, in line 1. In the second line, the "for" loop condition is checked $n+1$ *times with a cost of 2, thus* $2n+2$ *times in total. In the third line, addition and assignment operations will be repeated* n *times because of the "for" loop. The total time cost to execute the given algorithm is found as follows:*

$$1 + 2(n+1) + 2n = 4n + 3. \tag{2.17}$$

If we approximate the total time cost using the Big-O notation, we compute the time complexity of the algorithm as $\mathcal{O}(n)$.

In the given algorithm, we assume the input array has only integer values and the size of integer data type is four-byte for the processor used. We have an input array that has n *elements. Therefore, the input array allocates* $4n$ *bytes in the memory. Moreover, we have integer variables named sum,* i *and* n *each occupying four bytes in the memory. Thus, the algorithm's total space allocated in the memory is* $4n + 12$. *Accordingly, we present the space complexity as* $\mathcal{O}(n)$.

2.6. Summary of the Chapter

In this chapter, we briefly presented some of the graph theoretic concepts in the literature. These concepts are vital to the understanding of distributed algorithms and they will be utilized throughout this thesis. We also discussed two pioneering centralized methods to solve linear algebraic equations. Then, we reviewed several distributed algorithms for solving linear algebraic equations proposed in the literature. We also briefly mentioned about the fault-tolerant distributed consensus literature. Lastly, we reviewed algorithm complexity analysis methods regarding time spending and space allocation.

3. DETECTION OF FAULTY AGENTS IN THE NETWORK

In this chapter, we present four fault model definitions to analyze the impacts of a faulty agent on the networks. Then, we propose procedures to detect the different types of faulty behaviors in the network. We provide the necessary condition for the equation determination method which is crucial for fault detection. Moreover, we discuss the determination of linear equations in continuous-time systems.

3.1. Fault Models

Numerous fault model definitions for networks are introduced in the literature. However, the fault classifications may differ depending on the security needs in applications. For example, two sub-modes for faulty agents have been defined in [37, 38]: *completely arbitrary mode* and *Byzantine mode*. The Byzantine mode was first suggested by Lamport and originated from the informal problem statement explaining Byzantine generals' inconsistent behavior in a battlefield scenario [10, 38]. On the other hand, we consider the fault model classifications defined in [39] in this thesis. We briefly explain the main types of fault models depicted in [39] as follows:

- *Benign faults* are globally diagnosable by all normal agents in the system.
- *Symmetric faults* can deliver similar erroneous data to all receiving agents.
- *Asymmetric (Byzantine) faults* can transmit any form of erroneous data to all receiving agents.

Moreover, some studies suggest the hybrid fault models that are any mixture of the previously defined fault models can be experienced in real-world systems [40]. In other words, different severity levels might coexist in the system in the presence of hybrid faults.

Nevertheless, above definitions refer to a vast scope in fault model definitions. Thus, we extend the fault models introduced above to suggest more reliable fault-tolerant algorithms. Specifically, this thesis will focus on the following types of fault models:

- *Random-state Fault*: an agent that belongs to this fault type delivers the same arbitrary information to all its neighboring agents at all event times.
- *Fixed-state Fault*: an agent that belongs to this fault type sends a constant information to all its neighboring agents at all event times.
- *Single-faced Fault*: an agent that belongs to this fault type transmits an erroneous data generated in accordance with its function to all its neighboring agents at all event times.
- *Double-faced Fault*: an agent that belongs to this fault type sends different erroneous data generated in accordance with a particular function to each of its neighbors at all event times.

As mentioned before, we consider that all faults are categorized into three modes in this thesis. Therefore, we categorize the random-state, fixed-state and single-faced faults as sub-modes in the Symmetric fault model while the double-faced fault as a sub-mode in the Asymmetric fault model. The more rigorous definitions of above fault models are provided in the sequel.

Definition 3.1. (Random-state faulty agent) *A random-state faulty agent $j \in \mathcal{V}$ sends arbitrary state vectors $\boldsymbol{x}_j^i(t_j)$ to each neighboring agent i at each event time t_j.*

Example 3.1. *A random-state faulty agent broadcasts arbitrarily chosen state vectors to its neighboring agent i at each event time as illustrated in Figure 3.1.*

Definition 3.2. (Fixed-state faulty agent) *A fixed-state faulty agent $j \in \mathcal{V}$ publishes a constant state $\boldsymbol{x}_j^i(t_j)$ to each neighboring agent i at each event time t_j, i.e.,*

$$\boldsymbol{x}_j^i(1) = \boldsymbol{x}_j^i(2) = \cdots = c, \text{ where c} \in \mathbb{R}^n. \tag{3.1}$$

Example 3.2. *A fixed-state faulty agent $j \in \mathcal{V}$ sends the same state vector to its neighbors at each event time t_j as illustrated in Figure 3.2.*

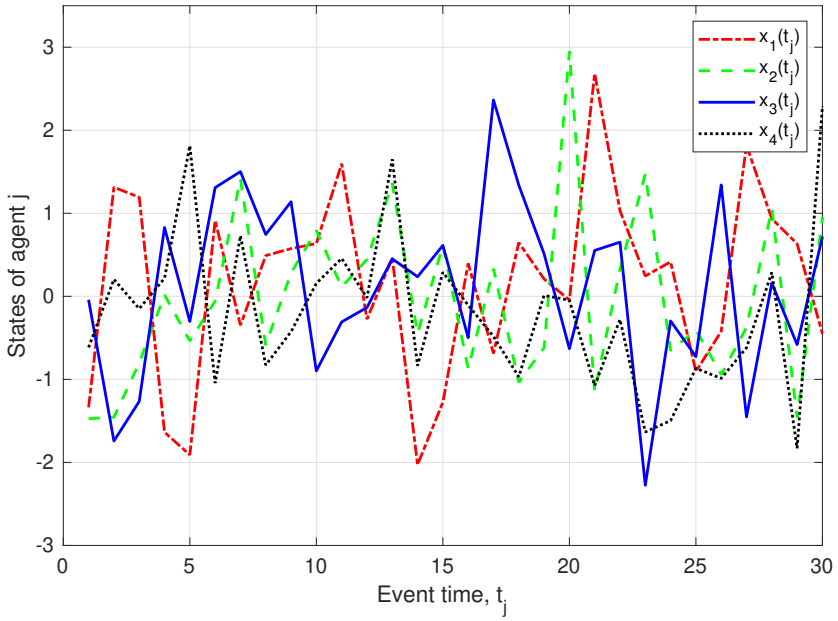


Figure 3.1. States of a random-state faulty agent

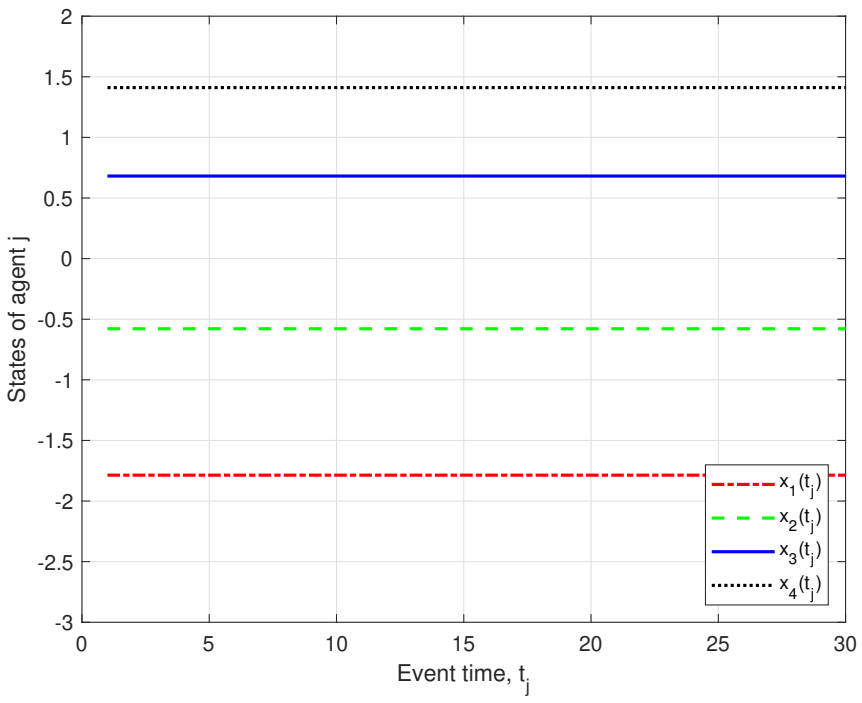


Figure 3.2. States of a fixed-state faulty agent

Definition 3.3. (Single-faced faulty agent) *A single-faced faulty agent $j \in \mathcal{V}$ sends the same solution estimate $\boldsymbol{x}_j^i(t_j)$ to each neighboring agent i at each event time t_j. Moreover, the state $\boldsymbol{x}_j^i(t_j)$ is a solution to some function $f_j(\cdot)$ that the normal agents do not seek to solve originally.*

Example 3.3. *Assume that a single-faced faulty agent j seeks to solve the equation of*

$$9x_1 - 6x_2 + 8x_3 - 4x_4 = 12. \tag{3.2}$$

It sends the solution estimates illustrated in Figure 3.3 and the first five solution estimates are provided in Table 3.1. It should be noted that the solution estimates presented in Table 3.1 are approximated in MATLAB environment with truncation errors.

Table 3.1. Solution estimates of a single-faced faulty agent.

Event time, t_j	**State Vectors**
1	$[1.1197, -0.7169, -0.0783,\ 0.4381]^T$
2	$[0.2795, -1.6102,\ 0.6004.\ 1.2450]^T$
3	$[0.9825, -0.8628,\ 0.0325,\ 0.5698]^T$
4	$[1.7052, -0.0944,\ 0.5513,\ 0.1243]^T$

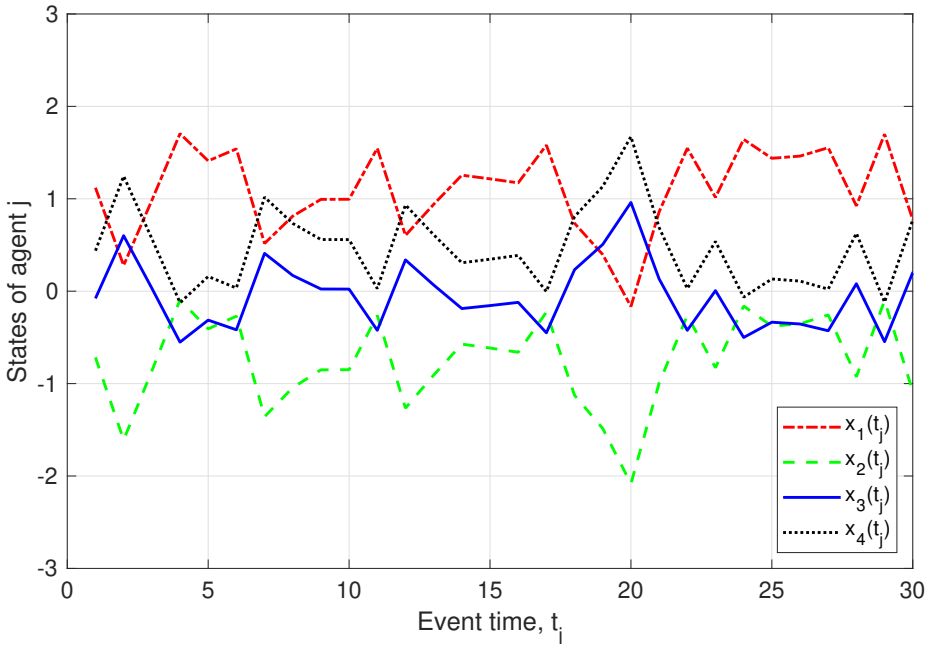


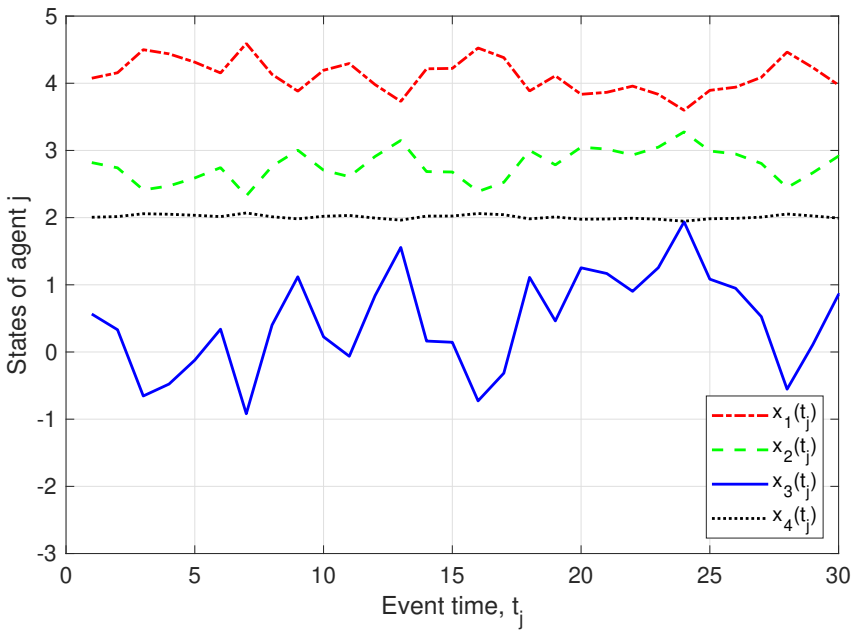
Figure 3.3. States of a single-faced faulty agent

Definition 3.4. (Double-faced faulty agent) *A double-faced faulty agent $j \in \mathcal{V}$ transmits different state vectors $\boldsymbol{x}_j^i(t_j)$ to each neighboring agent i at each event time t_j. It produces a state of $\boldsymbol{x}_j^i(t_j)$ in accordance with a different function $f_j^i(\cdot)$ defined for each neighboring agent i.*

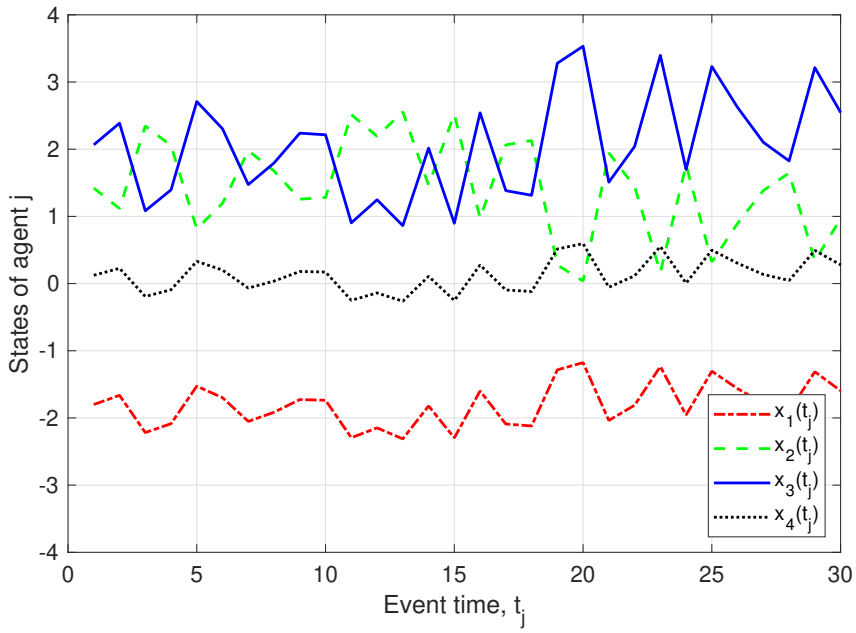
Example 3.4. *Suppose that a double-faced agent j sends its state vectors to its three neighboring agents at each event time t_j. The double-faced agent j owns a different equation set to solve for each neighbor, as given in Table 3.2. The state vectors that are feasible solutions to the specific equations given in Table 3.2 are illustrated in Figure 3.4.*

Table 3.2. The equations to be solved by the double-faced faulty agent j for each neighboring agent i.

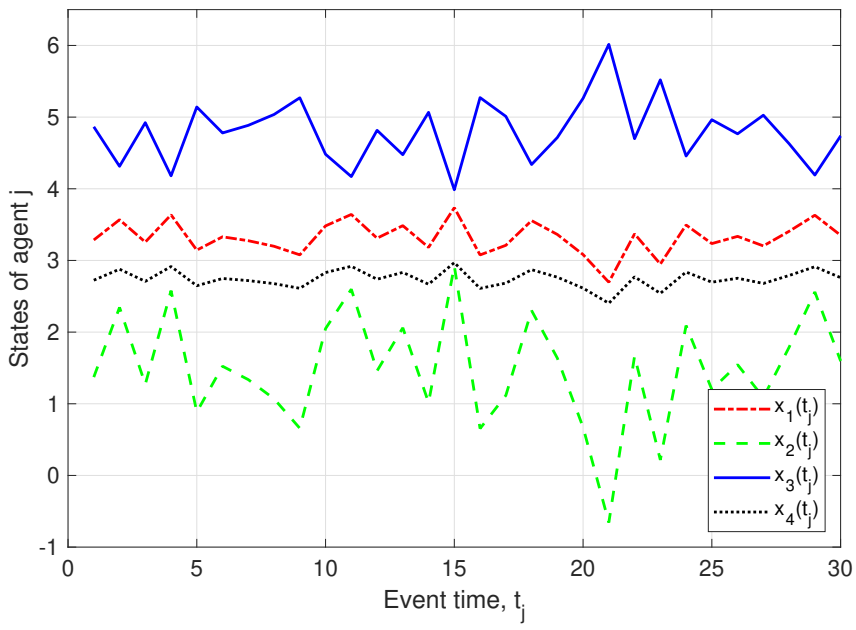
Equation, $f_j^i(\cdot)$	**Target Agent**, i
$6x_1 + 6x_2 + 7x_3 + 3x_4 = 117$	Agent 1
$-6x_1 + 4x_2 + 6x_3 + x_4 = 29$	Agent 2
$8x_1 - 4x_2 - x_3 + 7x_4 = 35$	Agent 3



(a)



(b)



(c)

Figure 3.4. States that double-faced faulty agent sends to a) neighbor Agent 1, b) neighbor Agent 2, c) neighbor Agent 3

3.2. Fault Detection

In the previous section, we introduced four fault models that can be observed in faulty networks. Therefore, normal agents should be equipped to eliminate the misbehaving effects of these faulty agents to achieve consensus on a solution to (1.1). In this section, we propose fault detection procedures that any normal agent can use to determine the intentions of its neighboring agents and cut ties with the detected misbehaving neighbors.

Due to security and privacy concerns, most current literature on solving linear algebraic equations in distributed systems presume that agents are not authorized to share their equation sets, i.e., $A_i\boldsymbol{x} = b_i$, with their neighbors. At first glance, this assumption may seem reasonable. However, an agent can deduce a linear equation system from received linearly independent solution estimates.

In Chapter 1, we mentioned that an equation system can be represented with the form of (1.1). However, this system may include linearly dependent equations and this may increase the complexity of the system. Therefore, we suggest a method to determine a system with r linearly independent equations by using linearly independent solutions of the original system. There are $n-r+1$ linearly independent solutions in the complete solution set of a non-homogeneous system $A\boldsymbol{x} = b$ where $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{x} \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ with $rank(A) = r$ and $rank([A\ b]) = r$, $(r < n)$ [41]. Then we realized that, this equation determination process may be implemented to a distributed algorithm to identify the misbehaving agents. Thus, we propose that an agent can deduce an equation system of its neighboring agent after receiving $n - r + 1$ linearly independent solution estimates from its neighbor. We indicated that the deduced equation system is equivalent to the system of the neighboring agent.

However, there are $n-r$ linearly independent solutions in the complete solution set of a homogeneous system [41], we require to receive $n-r$ linearly independent solutions to deduce an equation system which is equivalent to the system of neighboring agent.

In order to utilize this fact in fault detection phases, we introduce the following assumptions on each agent. These assumptions ensure that the necessary conditions for the detection phases are satisfied for each agent in the network.

Assumption 3.1. *Each agent i solves a system of the form $A_i\boldsymbol{x} = b_i$ and transmits*

(i) $n - r_i + 1$ *linearly independent solutions to its neighbors in a finite time interval, if* $b_i \neq \mathbf{0}_r$,

(ii) $n - r_i$ *linearly independent solutions to its neighbors in a finite time interval, if* $b_i = \mathbf{0}_r$.

In order to study fault detection algorithms, we also make the following assumptions on each agent in the network.

Assumption 3.2. *Each agent j has information on*

(i) whether $b_i = \mathbf{0}_n$ *or* $b_i \neq \mathbf{0}_n$, $\forall\ i \in \mathcal{N}_j$, *and*

(ii) the number of linearly independent equations, r_i, $\forall\ i \in \mathcal{N}_j$.

As indicated in Assumption 3.1, the system type determines the maximum number of linearly independent solutions to be deduced from a system of equations. Since each agent requires to know the system type of its neighboring agent to be able to specify the total number of linearly independent solutions received from its neighbor, we introduced Assumption 3.2. These assumptions are also required to analyze the complexity of the proposed algorithms in the next chapters.

Assumption 3.2 implies that each agent knows whether its neighbor tries to solve a homogeneous or a non-homogeneous system. This assumption is required because the maximum number of linearly independent solutions changes according to the system's type as indicated in Assumption 3.1. In addition, Assumption 3.2 helps to determine the number of linearly independent solutions to be taken from neighboring agents.

3.2.1. Detection of Random-state Faults

A random-state faulty agent i broadcasts an arbitrarily chosen state vector $\boldsymbol{x}_i \in \mathbb{R}^n$ to its neighboring agents at each event time. The main idea for detecting this type of misbehavior in the network is realizing that the randomized solution estimates cannot be a feasible solution to any equation. Therefore, the only equation that can satisfy the randomized states is the zero equation set, i.e.,

$$\begin{bmatrix} A_i & b_i \end{bmatrix} = \mathbf{0}_{r_i \times (n+1)} \tag{3.3}$$

where $A_i \in \mathbb{R}^{r_i \times n}$, $b_i \in \mathbb{R}^{r_i}$, and $\mathbf{0}_{r_i \times (n+1)}$ denotes the zero matrix.

Proposition 3.1. *Each agent j can deduce a linear equation system with the same solution set as system of neighboring agent i from $n - r_i + 1$ linearly independent solution estimates as stated in Section 3.2.*

Above proposition implies that even if an agent publishes $n - r_i + 1$ linearly independent solution estimates to its neighboring agent, all the other solution estimates should also validate the equation system deduced from the linearly independent estimates. Therefore, neighboring agent should receive at least $n - r_i + 2$ solution estimates from which $n - r_i + 1$ are linearly independent, in order to identify the random-state faulty agents.

Theorem 3.2. *Suppose agent i is a random-state faulty agent. Then the only equation set that satisfies the solutions of agent i is*

$$A_i \boldsymbol{x} = \mathbf{0}_{r_i} \tag{3.4}$$

with $A_i = \mathbf{0}_{r_i \times n}$.

Proof. Let $C_i \in \mathbb{R}^{n-r_i+1 \times n+1}$ be defined as

$$C_i = \begin{bmatrix} \boldsymbol{x}_i(1)^T & -1 \\ \boldsymbol{x}_i(2)^T & -1 \\ \vdots & \vdots \\ \boldsymbol{x}_i(n-r_i+1)^T & -1 \end{bmatrix}_{(n-r_i+1)\times(n+1)} \tag{3.5}$$

where $\boldsymbol{x}_i(t_i) \in \mathbb{R}^n$ $(t_i = 1, 2, \ldots, n - r_i + 1)$ are linearly independent solutions to system $A_i\boldsymbol{x} = b_i$. Then, we have the following:

$$C_i. \begin{bmatrix} \alpha_{i1} \\ \alpha_{i2} \\ \vdots \\ \alpha_{in} \\ \beta_i \end{bmatrix} = 0 \tag{3.6}$$

where $\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in}$ are the coefficients and β_i is the constant of ith equation of an equivalent system to system $A_i\boldsymbol{x} = b_i$.

We know that all solution estimates of agent i should satisfy the deduced equation system if it is a normal agent. Therefore, the neighboring agent can detect the misbehavior of agent i after receiving another solution estimate from agent i since the last solution estimate does not satisfy the deduced equation set for agent i. This yields that the equation set of agent i is the zero system, i.e., $[A_i \; b_i] = \mathbf{0}_{r_i \times (n+1)}$. □

3.2.2. Detection of Fixed-state Faults

As described previously, an agent i is known to be a fixed-state faulty agent if it publishes a constant vector in $\mathbb{R}^n$ to its neighbors for all event times.

Intuitively, if an agent i sends the same state vector $\boldsymbol{x}_i^j \in \mathbb{R}^n$ to its neighboring agent j at each event time t_i, this state can either be a faulty state or the unique solution to $A\boldsymbol{x} = b$. Nevertheless, the unique solution can be achieved only in two situations:

(i) all agents achieved a consensus on the unique solution to $A\boldsymbol{x}_i^j = b$ or
(ii) agent i is aware of the whole system of equations, i.e., $A_i = A$ and $b_i = b$.

However, we cannot validate the former since only the asymptotic convergence is guaranteed for the agents in algorithms (2.10) and (2.12). Likewise, the latter con-

tradicts to our assumption in Section 2.3 that indicates that none of the agents know the whole system of equations. Moreover, any agent can verify the uniqueness of the solution it receives by implementing it to its equation system since unique solution must be the solution that solves each private equation defined for agents.

In the sequel, we provide a theorem to examine the characteristics of fixed-state faulty agents.

Theorem 3.3. *Suppose agent i is a fixed-state faulty agent. Since agent i transmits constantly the same solution estimates in consecutive time steps, the deduced equation set of agent i will have a unique solution. Therefore, the projection matrix onto the kernel of the agent i's equation set can be computed as the zero matrix.*

Proof. Without loss of generality, agent j receives solution estimates of agent i for bounded $\mathbf{B} \geq n - r_i + 1$ time steps. We know that all solution estimates of agent i are the same for the fixed-state faulty agents, i.e.,

$$\boldsymbol{x}_i^j(1) = \boldsymbol{x}_i^j(2) = \ldots = \boldsymbol{x}_i^j(\mathbf{B}) = c \in \mathbb{R}^n. \tag{3.7}$$

Then we let $C_i \in \mathbb{R}^{\mathbf{B}\times n+1}$ be denoted as

$$C_i = \begin{bmatrix} c & -1 \\ c & -1 \\ \vdots & \vdots \\ c & -1 \end{bmatrix}_{\mathbf{B}\times(n+1)} . \tag{3.8}$$

Since all rows of C_i are the same, we have $rank(C_i) = 1$. From Rank-Nullity Theorem [2], we have the following

$$rank(C_i) + dim(null(C_i)) = n + 1 \tag{3.9}$$

which yields $dim(null(C_i)) = n$. Note that $null(C_i)$ span linearly independent vectors that indicate the equation system of agent i as $[A_i\ b_i]$. This implies that $rank(A_i) = n$. Again from Rank-Nullity Theorem, we have

$$rank(A_i) + dim(null(A_i)) = n. \tag{3.10}$$

Since this indicates $dim(null(A_i)) = 0$, we conclude the projection matrix onto the

kernel of A_i is the zero matrix, i.e., $P_{ker(A_i)} = \mathbf{0}_n$. □

3.2.3. Detection of Single-faced Faults

In this section, we analyze a detection condition for single-faced faulty agents. As mentioned in the previous section, agent i is known to be a *single-faced faulty agent* if it shares the same erroneous data with each neighboring agent.

We consider that each agent utilizes the synchronous distributed algorithm proposed in [22]. On the other hand, results can be directly extended to asynchronous distributed algorithm developed in [24]. The underlying graph is defined as repeatedly jointly strongly connected for asymptotic convergence in algorithm (2.10).

We suppose that each agent i sends two vectors to its neighboring agent j at each event time: its solution estimate $\boldsymbol{x}_i^j(t_i)$ and the average of received solution estimates $\boldsymbol{d}_i(t_i)$. In addition, each normal agent j in network $\mathcal{G}$ knows only its own equation set $A_j\boldsymbol{x} = b_j$. Thus, each agent j should accept all data it receives from agent i to assess its reliability. We provide the following theorem to detect the misbehavior of a single-faced faulty agent.

Theorem 3.4. *Assume agent i is a single-faced faulty agent. Then, we have*

$$\boldsymbol{x}_i^j(t_i) - \boldsymbol{x}_i^j(t_i - 1) \neq -P_{ker(A_i)}\big(\boldsymbol{x}_i^j(t_i - 1) - \boldsymbol{d}_i(t_i - 1)\big) \tag{3.11}$$

where $\boldsymbol{x}_i(t_i)$ is the solution estimate of agent i at event time t_i, $\boldsymbol{x}_i(t_i - 1)$ is the solution estimate of agent i at event time $(t_i - 1)$, $\boldsymbol{d}_i(t_i - 1)$ is the average of received state vectors from in-neighbors of agent i, i.e., $\boldsymbol{d}_i(t_i - 1) = \frac{1}{\mu_i}\sum_{s\in\mathcal{N}_i^+}\boldsymbol{x}_s(t_i - 1)$, and $P_{ker(A_i)}$ is the projection matrix onto the kernel of the equation set of agent i.

Proof. Without loss of generality, agent j forms C_i for its neighboring agent i as

$$C_i = \begin{bmatrix} \boldsymbol{x}_i(1)^T & -1 \\ \boldsymbol{x}_i(2)^T & -1 \\ \vdots & \vdots \\ \boldsymbol{x}_i(n-r_i+1)^T & -1 \end{bmatrix}_{(n-r_i+1)\times(n+1)} \tag{3.12}$$

where $\boldsymbol{x}_i(t_i) \in \mathbb{R}^n$ $(t_i = 1, 2, \ldots, n - r_i + 1)$ are linearly independent solutions to system $A_i\boldsymbol{x} = b_i$. Let $u_i = [\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in}, \beta_i]^T \in \mathbb{R}^{n+1}$ where $\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in}$ are the coefficients and β_i is the constant of the general form of ith equation of an equivalent system to system $A_i\boldsymbol{x} = b_i$. Then, we have

$$C_i u_i = 0. \tag{3.13}$$

From the fact given in Section 3.2, agent j can deduce an equation set with the same solution set as system $A_i\boldsymbol{x} = b_i$ from the null space of matrix C_i. Then, agent j computes the projection matrix of agent i, $P_{ker(A_i)}$, by using (2.11).

Since each agent in $\mathcal{G}$ utilizes the same distributed algorithm (2.10) in order to update its solution estimate, the following equality must be satisfied for agent i:

$$\boldsymbol{x}_i^j(t_i) - \boldsymbol{x}_i^j(t_i - 1) = -P_{ker(A_i)}\big(\boldsymbol{x}_i^j(t_i - 1) - \boldsymbol{d}_i(t_i - 1)\big). \tag{3.14}$$

On the other hand, this equality does not hold for a single-faced faulty agent since the average of received neighboring estimates $\boldsymbol{d}_i(t_i - 1)$ does not satisfy the required update on consecutive time steps of faulty agent i. □

Corollary 3.5. *A single-faced faulty agent i can be detected when the following condition holds:*

$$\begin{aligned} ||\boldsymbol{h}_i(t_i)|| &= \big|\big|\boldsymbol{x}_i^j(t_i - 1) - \boldsymbol{x}_i^j(t_i) - P_{ker(A_i)}\big(\boldsymbol{x}_i^j(t_i - 1) - \boldsymbol{d}_i(t_i - 1)\big)\big|\big| \\ &\neq 0 \end{aligned} \tag{3.15}$$

where $||\cdot||$ denotes the norm of its argument.

So far, we have assumed that the underlying topology was repeatedly jointly strongly connected. Therefore, we require each agent to deliver two separate data at each event time for verifying algorithm (2.10). On the other hand, it will be sufficient

for each agent to share only its solution estimate with its neighbors when the underlying graph of the network is *complete*.

Corollary 3.6. *For complete graphs, each agent $i \in \mathcal{V}$ is directly connected to every other agent j. Therefore, we have*

$$\boldsymbol{d}_i(t_i - 1) = \boldsymbol{d}_j(t_i - 1) \tag{3.16}$$

for all $t_i = 1, 2, \ldots$. Thus, agent j does not necessarily have to receive $\boldsymbol{d}_i(t_i - 1)$ from agent i.

3.2.4. Detection of Double-faced Faults

We assume that each agent utilizes the averaging-based distributed consensus algorithm proposed in [22] to estimate a solution for its equation set. Therefore, the underlying graph of the network is needed to be repeatedly jointly strongly connected, as stated in algorithm (2.10).

Each agent $i \in \mathcal{V}$ should transmit two separate vectors to its neighbors for all $t_i = 1, 2, \ldots$: estimated solution vector, $\boldsymbol{x}_i^j(t_i)$, and the average of received state vectors of its neighboring agents, $\boldsymbol{d}_i(t_i)$. As described in the previous section, a *double-faced faulty agent i* forms different solution estimate vectors, $\boldsymbol{x}_i^j(t_i)$, for each agent $j \in \mathcal{N}_i^-$ for all $t_i = 1, 2, \ldots$. Besides, each solution estimate should be a feasible solution to a distinct equation set.

As previously stated in Section 3.2, each agent j can deduce an equation system with the same solution set as equation set of agent i from $n - r_i + 1$ linearly independent solutions. Furthermore, agent j considers only the state vectors received from agent i to identify its behavior is whether faulty. To do this, agent j can directly use the condition proposed in Theorem 3.4.

Lemma 3.7. *Assume agent i is a double-faced faulty agent. Then, we have*

$$\boldsymbol{x}_i^j(t_i) - \boldsymbol{x}_i^j(t_i - 1) \neq -P_{ker(A_i)}\big(\boldsymbol{x}_i^j(t_i - 1) - \boldsymbol{d}_i(t_i - 1)\big) \tag{3.17}$$

where $\boldsymbol{x}_i(t_i)$ is the solution estimate of agent i at event time t_i, $\boldsymbol{x}_i(t_i-1)$ is the solution estimate of agent i at event time (t_i-1), $\boldsymbol{d}_i(t_i-1)$ is the average of received state vectors from in-neighbors of agent i, i.e., $\boldsymbol{d}_i(t_i-1)=\frac{1}{\mu_i}\sum_{s\in\mathcal{N}_i^+}\boldsymbol{x}_s(t_i-1)$, and $P_{ker(A_i)}$ is the projection matrix onto the kernel of the equation set of agent i.

Proof. The proof directly follows from Theorem 3.4. □

3.2.5. Numerical Analysis of Fault Detection

In this section, we present numerical examples to analyze the validity of proposed fault detection approaches.

Example 3.5. (Detection of Random-state faults) *We assume that normal agent $j\in\mathcal{V}$ receives solution estimates from agent $i\in\mathcal{N}_j^+$. Table 3.3 presents five different solution estimates that agent j received from its neighboring agent i.*

Table 3.3. Solution estimates of random-state faulty agent i.

Event time, t_i	**State Vectors**
1	$[0.5377, -1.3077, -1.3499, -0.2050]^T$
2	$[1.8339, -0.4336,\ 3.0349, -0.1241]^T$
3	$[-2.2588,\ 0.3426,\ 0.7254,\ 1.4897]^T$
4	$[0.8622,\ 3.5784, -0.0631,\ 1.4090]^T$
5	$[0.3188,\ 2.7694,\ 0.7147,\ 1.4172]^T$

Normal agent j forms C_i as

$$C_i=\begin{bmatrix} 0.5377 & -1.3077 & -1.3499 & -0.2050 & -1\\ 1.8339 & -0.4336 & 3.0349 & -0.1241 & -1\\ -2.2588 & 0.3426 & 0.7254 & 1.4897 & -1\\ 0.8622 & 3.5784 & -0.0631 & 1.4090 & -1\\ 0.3188 & 2.7694 & 0.7147 & 1.4172 & -1 \end{bmatrix}. \tag{3.18}$$

Then, agent j computes $null(C_i)$ and finds

$$dim(null(C_i)) = 0 \tag{3.19}$$

This result indicates that $A_i = \mathbf{0}_{r_i \times n}$. Therefore, agent j labels agent i as a random-state faulty agent.

Example 3.6. (Detection of Fixed-state faults) *We assume an agent $i \in \mathcal{V}$ publishes the constant state vector*

$$[1.6285, -0.8712,\ 2.5237, -1.3152]^T \tag{3.20}$$

to its out-neighbor agent j at each event time. Then, agent j generates C_i as

$$C_i = \begin{bmatrix} 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \\ 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \\ 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \\ 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \\ 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \\ 1.6285 & -0.8712 & 2.5237 & -1.3152 & -1 \end{bmatrix}. \tag{3.21}$$

Then, agent j computes $null(C_i)$ and finds

$$dim(null(C_i)) = n \tag{3.22}$$

which implies $dim(null(A_i)) = 0$. Since, the projection matrix onto the kernel of A_i can be computed as zero matrix, we conclude that agent i is indeed a faulty agent.

Example 3.7. (Detection of single-faced & double-faced faults) *Let agent i in $\mathcal{G}$ sends its state vectors of $\boldsymbol{x}_i^j(t_i)$ and $\boldsymbol{d}_i(t_i)$ to its out-neighbor agent j at each event time t_i as shown in Table 3.4.*

Table 3.4. Solution estimates and average states of neighbors of agent i

t_i	**Solution Estimates, $\boldsymbol{x}_i^j(t_i)$**	**Av. States of Neighbors of $\boldsymbol{i}$, $\boldsymbol{d}_i(t_i)$**
1	$[0.6660,\ 1.5821,\ 3.0333,\ 4.1869]^T$	$[1.0638,\ 2.4597,\ 1.7143,\ 2.6093]^T$
2	$[0.9255,\ 1.7645,\ 3.0910,\ 4.0349]^T$	$[1.2473,\ 2.6032,\ 2.2180,\ 3.1990]^T$
3	$[0.9165,\ 1.3252,\ 3.3433,\ 4.0196]^T$	$[1.2536,\ 2.3101,\ 2.4502,\ 3.3920]^T$
4	$[0.9182,\ 0.9177,\ 3.5837,\ 3.9991]^T$	$[1.2965,\ 2.2985,\ 2.5909,\ 3.5079]^T$

Table 3.4. Solution estimates and average states of neighbors of agent i (cont.)

5	$[0.9185,\ 0.6412,\ 3.7463,\ 3.9857]^T$	$[1.3847,\ 2.6041,\ 2.6372,\ 3.5569]^T$
6	$[0.9926,\ 1.1326,\ 3.5047,\ 3.9633]^T$	$[1.4116,\ 2.7276,\ 2.6138,\ 3.5313]^T$

Then, agent j forms C_i as

$$C_i = \begin{bmatrix} 0.6660 & 1.5821 & 3.0333 & 4.1869 & -1 \\ 0.9255 & 1.7645 & 3.0910 & 4.0349 & -1 \\ 0.9165 & 1.3252 & 3.3433 & 4.0196 & -1 \\ 0.9182 & 0.9177 & 3.5837 & 3.9991 & -1 \\ 0.9185 & 0.6412 & 3.7463 & 3.9857 & -1 \\ 0.9926 & 1.1326 & 3.5047 & 3.9633 & -1 \end{bmatrix}. \tag{3.23}$$

Next, agent j computes $null(C_i)$ as

$$null(C_i) = \left\{ \alpha_1 \begin{bmatrix} 0.6136 \\ -0.3303 \\ -0.5258 \\ 0.4519 \\ 0.1833 \end{bmatrix} + \alpha_2 \begin{bmatrix} 0.0939 \\ -0.1160 \\ -0.2021 \\ -0.0555 \\ -0.9663 \end{bmatrix} : \alpha_1, \alpha_2 \in \mathbb{R} \right\} \tag{3.24}$$

which implies the following linear equation set

$$\begin{aligned} -2x_1 + 5x_2 + 9x_3 + 6x_4 &= 77 \\ 5x_1 - 2x_2 - 3x_3 + 5x_4 &= 17. \end{aligned} \tag{3.25}$$

Moreover, agent j computes the projection matrix of agent i by using (2.11) *as*

$$P_{ker(A_i)} = \begin{bmatrix} 0.6002 & 0.1793 & 0.2758 & -0.3630 \\ 0.1793 & 0.7958 & -0.3536 & -0.0731 \\ 0.2758 & -0.3536 & 0.3828 & -0.1876 \\ -0.3630 & -0.0731 & -0.1876 & 0.2212 \end{bmatrix} \tag{3.26}$$

utilizing the determined equation set of agent i. Agent j examines Corollary 3.5 to identify the intention of agent i. Then agent j finds

$$||\boldsymbol{h}_i(t_i)|| = 0.6352 \neq 0. \tag{3.27}$$

This result indicates that agent i is a faulty agent.

3.3. Extension to Continuous-time Systems

So far, we have assumed that each agent utilizes discrete-time distributed algorithms in fault detection schemes to identify faulty neighbors. Therefore, we know that each agent shares its state vectors in discrete-time steps. On the other hand, proposed fault detection schemes can directly be integrated into continuous-time distributed algorithms.

We have shown that each agent can deduce an equivalent equation system to the neighboring agent's system provided that sufficient number of solutions to the equation system is received. Therefore, we have proposed the idea that normal agents can identify all neighboring faulty agents by examining the criteria regarding faulty models' characteristics.

In continuous-time systems, the distributed problem of solving linear algebraic equations can be represented by a differential equation of the form (2.14) [7, 33, 34]. Thus, we conclude that each agent solves its system of equations using continuous-time solution estimates of neighboring agents. On the other hand, the fault detection schemes require $n - r + 1$ linearly independent solution estimates for the equation system determination process. Therefore, each agent should send its state vectors to its neighbors in line with a sampling time, $\tau_s > 0$. In this way, each agent can collect $n - r + 1$ linearly independent solution estimates from its neighbors in a bounded time interval.

3.4. Summary of the Chapter

We know that each agent in the network should be non-faulty for achieving consensus by using an averaging-based distributed algorithm. If any intrusion of faulty agents occurs into the network, faulty agents may prevent the consensus among the normal agents unless the normal agents detect the faulty ones. Therefore, it is crucial to eliminate the effects of faulty agents to achieve consensus.

In this chapter, we have introduced four fault models that may be experienced in multi-agent networks:

(i) Random-state fault model
(ii) Fixed-state fault model
(iii) Single-faced fault model
(iv) Double-faced fault model.

These fault models originate from the fault types defined in [39], and we can consider the first three fault models as Symmetric faults while the last model is an Asymmetric (Byzantine) fault. We consider that each of the fault models has its own characteristics. In order to eliminate the effects of faulty agents, we have introduced our fault detection schemes for each fault model. Furthermore, we have provided simulation results for the detection phase of each fault model. We also discussed equation determination process in continuous-time systems.

4. SYNCHRONOUS FAULT RESILIENT DISTRIBUTED ALGORITHM

In Chapter 3, we have introduced four different faulty agent models and the detection algorithms to isolate these misbehaving agents from the network. In this chapter, we propose a synchronous fault resilient discrete-time distributed algorithm to solve linear algebraic equations in faulty networks. Secondly, we analyze the time and space complexities of the suggested algorithm. Later on, we present a numerical example to illustrate the theoretical results.

4.1. The Synchronous Algorithm

Consider a time-varying network $\mathcal{G}(t) = (\mathcal{V}, \mathcal{E}(t))$ with $m > 1$ autonomous agents. We also defined the set of nodes in the network as follows: $\mathcal{V} = \mathcal{V}_n \cup \mathcal{V}_f$ where $\mathcal{V}_n$ denotes the set of normal agents and $\mathcal{V}_f$ is the set of faulty agents in the network.

Initially, each agent i only knows its own equation set of the form $[A_i \ b_i]$ and is unaware of the equations owned by other agents. Moreover, each agent $i \in \mathcal{V}$ seeks to solve its own equation set with a synchronous discrete-time distributed algorithm, i.e., algorithm (2.10) proposed in [22]. Thus, the underlying topology for the network is repeatedly jointly strongly connected as introduced in [22]. In addition, it is known that each agent updates its solution estimate vectors and transmits them to its neighbors at synchronous event times. Furthermore, each agent i receives two vectors at each time-step t from agent $j \in \mathcal{N}_i^+$:

(i) the solution estimate at step-time t, i.e. $\boldsymbol{x}_j(t) \in \mathbb{R}^n$ and

(ii) the average of the solution estimates of the neighboring agents of the agent j, i.e.,

$$\boldsymbol{d}_j(t) = \frac{1}{\mu_j(t)} \sum_{s \in \mathcal{N}_j^+(t)} \boldsymbol{x}_s(t) \in \mathbb{R}^n \text{ where } \mu_j(t) = |\mathcal{N}_j^+(t)| \text{ and } t \geq 1. \tag{4.1}$$

We presume that all state vectors that agent i receives from its in-neighbor agents throughout the process are linearly independent. We stated in Section 3.2 that the agent i must receive $n - r_j + 1$ consecutive solution estimates from agent $j \in \mathcal{N}_i^+$ to determine an equation set with the same solution set as equation set of agent j.

4.2. Description of the Algorithm

In this section, we give a detailed description for the synchronous fault resilient discrete-time distributed algorithm illustrated in Figures 4.1 and 4.2.

Phase #0 : Agent i chooses its initial state with a specific vector which is a feasible solution to its own equation set. Then agent i computes the projection matrix using (2.11). In addition, agent i counts the number of agents in its in-neighbor list and initializes an empty matrix for each in-neighbor to be utilized in the next phases. We also set a constant small precision value ϵ_c for the simulations.

Phase #1 : In this phase, agent i listens to each of its neighboring agent j for $n - r_i + 1$ iterations and constructs C_j matrices for all its neighbors with the received solution estimates. Agent i utilizes matrix C_j to determine the equations of each neighboring agent j in the following phases.

Phase #2 : In this phase, agent i labels each neighboring agent j as either a normal or a faulty agent. We divide this phase into three sub-phases.

- In Phase #2.1, agent i computes a basis for the null space of C_j, which was constructed in Phase #1, and appends each basis vector into the columns of matrix M_j. As provided in Theorem 3.2, random-state faulty agents can be identified when the norm of the matrix M_j is equal to zero. If the given condition holds, agent i labels the corresponding agent $j \in \mathcal{N}_i^+$ as a random-state faulty agent and removes agent j from its in-neighbor set.

1: $\boldsymbol{x}_i(1) \leftarrow A_i^T(A_iA_i^T)^{-1}b_i$ ▶ Phase #0
2: $P_{A_i} \leftarrow I - A_i^T(A_iA_i^T)^{-1}A_i$
3: $\mu_i \leftarrow |\mathcal{N}_i^+|$
4: $\epsilon_c \leftarrow$ a small precision value
5: **foreach** $j \in \mathcal{N}_i^+$ **do**
6: Initialize C_j as an empty 2D array
7: **end for**
8: **for** $t \leftarrow 1$ **to** $n - r_j + 1$ **do** ▶ Phase #1
9: Initialize $\boldsymbol{d}_i(t)$ as an empty vector in $\mathbb{R}^n$
10: **foreach** $j \in \mathcal{N}_i^+$ **do**
11: $\boldsymbol{d}_j(t)$ received from j
12: $\boldsymbol{x}_j(t)$ received from j
13: $w_{ij} \leftarrow 1/\mu_i$
14: $\boldsymbol{d}_i(t) \leftarrow \boldsymbol{d}_i(t) + w_{ij}\boldsymbol{x}_j(t)$
15: Append the rows of $[\boldsymbol{x}_j(t)^T, \ -1]$ to C_j
16: **end for**
17: $\boldsymbol{x}_i(t+1) \leftarrow \boldsymbol{x}_i(t) - P_{A_i}\Big(\boldsymbol{x}_i(t) - \boldsymbol{d}_i(t)\Big)$
18: **end for**
19: **foreach** $j \in \mathcal{N}_i^+$ **do** ▶ Phase #2
20: $M_j \leftarrow$ a basis for the $null(C_j)$ ▷ Phase #2.1
21: **if** $||M_j||_1 \neq 0$ **then**
22: $\mathcal{N}_i^+ \leftarrow \mathcal{N}_i^+ \setminus j$
23: **continue**
24: **end if**

Figure 4.1. Synchronous fault resilient distributed algorithm for solving linear algebraic equations

$\bar{A}_j \leftarrow M_j[1:n][:]^T$ ▷ Phase #2.2

$\bar{P}_{\bar{A}_j} \leftarrow I - \bar{A}_j{}^T(\bar{A}_j\bar{A}_j{}^T)^{-1}\bar{A}_j$

if $||\bar{P}_{\bar{A}_j}||_1 < n^2\epsilon_c$ **then**

  $\mathcal{N}_i^+ \leftarrow \mathcal{N}_i^+ \setminus j$

  continue

end if

$\boldsymbol{h}_j \leftarrow \boldsymbol{x}_j(t) - \boldsymbol{x}_j(t-1) - \bar{P}_{\bar{A}_j}\Big(\boldsymbol{x}_j(t-1) - \boldsymbol{d}_j(t-1)\Big)$ ▷ Phase #2.3

if $||\boldsymbol{h}_j||_1 > n\epsilon_c$ **then**

  $\mathcal{N}_i^+ \leftarrow \mathcal{N}_i^+ \setminus j$

  continue

end if

end for

$\mu_i \leftarrow |\mathcal{N}_i^+|$ ▶ Phase #3

do

  Initialize $\boldsymbol{d}_i(t)$ as an empty vector in $\mathbb{R}^n$

  foreach $j \in \mathcal{N}_i^+$ **do**

    $w_{ij} \leftarrow 1/\mu_i$

    $\boldsymbol{d}_i(t) \leftarrow \boldsymbol{d}_i(t) + w_{ij}\boldsymbol{x}_j(t)$

  end for

  $\boldsymbol{x}_i(t+1) \leftarrow \boldsymbol{x}_i(t) - P_i\Big(\boldsymbol{x}_i(t) - \boldsymbol{d}_i(t)\Big)$

  $t \leftarrow t+1$

while $||\boldsymbol{x}_i(t+1) - \boldsymbol{x}_i(t)|| > \epsilon_c$

Figure 4.2. Synchronous fault resilient distributed algorithm for solving linear algebraic equations (cont.)

- In Phase #2.2, agent i determines the equations of each of its neighbor agent j. Then, agent i computes the projection matrix onto the kernel of equations of agent j. Thus, agent i can identify the fixed-state faulty agents by the condition defined in Theorem 3.3. Accordingly, agent i removes the faulty agent from $\mathcal{N}_i^+$ if this condition does not hold for the corresponding neighbor agent.
- Since the general detection algorithm for the single-faced and double-faced faulty agents are similar, agent i can simultaneously detect such faults in Phase #2.3. Firstly, agent i should compute the difference vector $\boldsymbol{h}_j$ as described in (3.15). Agent i checks the condition proposed in Theorem 3.4, i.e., $||\boldsymbol{h}_j(t_j)||_1 \neq 0$ to determine whether neighbor j is normal agent or not. If agent i labels agent j as a faulty agent, it removes agent j from its in-neighbor set.

Phase #3 : Until this phase, agent i has determined which of its neighboring agents are trustworthy and which are not. Thus, agent i updates the number of neighboring agents in accordance with the updated neighbor list. Since the updated neighbor list includes only the normal agents, agent i executes the discrete-time distributed algorithm (2.10) proposed in [22] to achieve consensus with the other normal agents.

4.3. Algorithm Complexity Analysis

In this section, we analyze the time and space complexities of the proposed algorithm illustrated in Figures 4.1 and 4.2.

We first consider the time complexity of the given algorithm phase by phase. In Phase #0, the state initialization takes $\mathcal{O}(r_j^2 n)$ time because of the matrix inversion method presented in [42]. Moreover, the projection matrix calculation requires $\mathcal{O}(r_j n^2)$ times. Therefore, the total time cost of Phase #0 is $\mathcal{O}(r_j n^2)$.

In Phase #1, line 13 takes $\mathcal{O}(n)$ time, but the total time required for this operation is $\mathcal{O}(n\mu_i)$ because of the "foreach" loop. Line 16 takes $\mathcal{O}(n^2)$ time due to the matrix and vector multiplications. Since we have a "for" loop, the total time required

for Phase#1 is $\mathcal{O}(n^3)$.

In Phase #2, a basis computation for the null space of an $(n - r_j + 1) \times (n + 1)$ matrix C_j takes $\mathcal{O}(n^3)$ for singular value decomposition (SVD) method suggested in [43].

- In Phase #2.1, we have a conditional statement in line 20 that computes the 1-norm of an $(n + 1) \times r$ matrix with the time cost of $\mathcal{O}(n)$ as described in [44].
- When we consider Phase #2.2, the projection computation takes $\mathcal{O}(n^2)$ times. Also, for the conditional statement in line 26, we compute the 1-norm of an $n \times n$ matrix that costs $\mathcal{O}(n^2)$ times.
- The difference vector computation in Phase #2.3 requires $\mathcal{O}(n^2)$ owing to the matrix and vector multiplications. The conditional statement in line 31 costs $\mathcal{O}(n)$ times due to the 1-norm computation of a vector of size $(n \times 1)$.

It should be noted that all operations in Phase #2 are repeated μ_i times because of the "foreach" loop. Therefore, Phase #2 requires $\mathcal{O}(n^3\mu_i)$ times in total.

Lastly, Phase #3 takes $\mathcal{O}(n^2)$ times in total by cause of the matrix and vector multiplication in line 41. On the other hand, we do not consider Phase #3 for the overall complexity analysis since this phase is not related to the fault detection algorithm. Hence, the total time required for the introduced phases is $\mathcal{O}(n^3\mu_i)$.

The space complexity of Phase #0 is $\mathcal{O}(n^2\mu_i)$ for the 2D array initialization for each neighbor. In Phase #1, we also have several $n \times 1$ sized vector assignments for each neighbor. Moreover, we have a "for" loop, which iterates $n - r_i + 1$ times. Therefore, the total space cost for Phase #1 is $\mathcal{O}(n^2\mu_i)$. In Phase #2, we have several vectors and 2D array assignments. Specifically, the projection matrix for each neighbor allocates $\mathcal{O}(n^2\mu_i)$ space in the memory. Accordingly, the total space complexity for the overall fault detection algorithm is $\mathcal{O}(n^2\mu_i)$.

4.4. A Numerical Example

Consider the faulty network $\mathcal{G}_f$ illustrated in Figure 4.3. There are eight agents in the network, and agents do not know the intentions of their neighbors. In addition, we assume that the internal clocks of each agent are perfectly synchronized, and each agent follows the synchronous discrete-time distributed algorithm (2.10) suggested in [22] after the fault detection phase. It should also be noted that each of the vectors presented in this example has a truncation error since we utilize the MATLAB environment for the simulations. Therefore, in this example, we set the precision value ϵ_c as 10^{-15}.

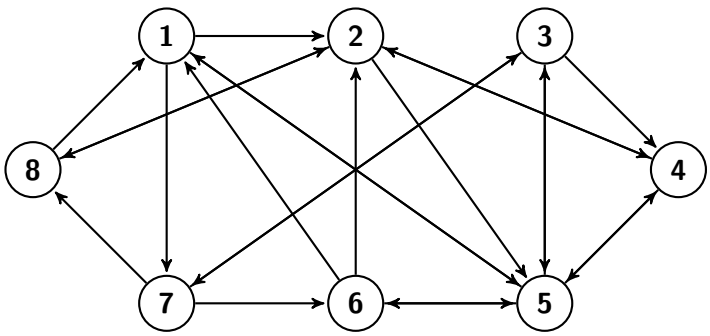


Figure 4.3. A faulty network $\mathcal{G}_f$.

Each agent $i \in \mathcal{V}$ shares two vectors in $\mathbb{R}^n$ with its neighboring agents at each event time t_i: its solution estimate and the average of received solution estimates of its neighbors, $\boldsymbol{x}_i(t_i)$ and $\boldsymbol{d}_i(t_i)$, respectively. The system of equations that normal agents seek to solve cooperatively is provided as follows:

$$\begin{aligned} -x_1 - 6x_2 - 7x_3 - 5x_4 &= -54 \\ -4x_1 + 8x_2 - 5x_3 + 2x_4 &= 5 \\ 8x_1 + 9x_2 - 2x_3 + 4x_4 &= 36 \\ -x_1 - x_2 + 2x_3 - 5x_4 &= -17. \end{aligned} \tag{4.2}$$

Each normal agent owns only one distinct equation in the given system. Furthermore, the system has a unique solution of $[1,\ 2,\ 3,\ 4]^T$. However, this is not known by the agents primarily.

In this example, we observe the procedure that Agent 5 follows to determine the intentions of its neighbors. We consider Agent 5 in $\mathcal{G}_f$ is a normal agent and the equation that Agent 5 seeks to solve is

$$8x_1 + 9x_2 - 2x_3 + 4x_4 = 36. \tag{4.3}$$

According to Phase #0 of the procedure illustrated in Figure 4.1, Agent 5 initializes its state vector as $[1.3631,\ 1.4851, -1.0181,\ 2.4231]^T$ by using the minimum norm solution method and computes its projection matrix by the equation given in (4.3). Also, Agent 5 counts the number of its in-neighbor agents and creates a 2D matrix C_j for each of its in-neighbor agents.

Then, Agent 5 proceeds to Phase #1 to collect sufficient data for the fault detection scheme. All data that Agent 5 received from its neighbors during the process, are given in Tables 4.1 and 4.2, respectively. Finally, agent 5 appends the received vectors to matrix C_j for each of its neighbors separately.

Table 4.1. Solution estimates of the neighbor agents of Agent 5, $\boldsymbol{x}_j(t)$.

Neighbor Agent, j	**Event time,** t	**Estimated Solution,** $\boldsymbol{x}_j(t)$
Agent 1	1	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	2	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	3	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	4	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	5	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
Agent 2	1	$[0.8355, -0.3428, -0.4780, -0.8891]^T$
	2	$[1.2634, 0.3832, -0.1189, 0.4172]^T$
	3	$[1.0132, -0.8695, -0.7947, 0.6885]^T$
	4	$[1.5857, 1.2502, -0.1156, -1.3318]^T$
	5	$[-2.3428, -0.9266,\ 1.1296, -0.5491]^T$

Table 4.1. Solution estimates of the neighbor agents of Agent 5, $\boldsymbol{x}_j(t)$ (cont.)

Agent 3	1	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	2	$[1.5377,\ 2.4908,\ 3.0490,\ 3.2348]^T$
	3	$[1.6652,\ 2.5273,\ 3.0033,\ 3.2296]^T$
	4	$[1.5658,\ 2.4891,\ 3.0320,\ 3.2552]^T$
	5	$[1.9867,\ 2.5850,\ 2.8634,\ 3.2918]^T$
Agent 4	1	$[-0.4762, -0.3968,\ 0.6349,\ 0.0794]^T$
	2	$[0.1010,\ 0.1853,\ 1.4382,\ 0.0266]^T$
	3	$[0.3732,\ 0.3050,\ 1.7203,\ 0.0018]^T$
	4	$[0.5135,\ 0.5294,\ 1.9625,\ 0.0277]^T$
	5	$[0.5394,\ 0.8314,\ 2.1646,\ 0.0764]^T$
Agent 6	1	$[1.0263,\ 0.6842,\ 1.3684, -1.0263]^T$
	2	$[1.2983,\ 1.0941,\ 1.2658, -0.6179]^T$
	3	$[1.2705,\ 0.9937,\ 1.4682, -0.4428]^T$
	4	$[1.2637,\ 1.1187,\ 1.5355, -0.2765]^T$
	5	$[1.0510,\ 1.2406,\ 1.7056, -0.1812]^T$

Table 4.2. Received state average vectors from the neighbor agents of Agent 5, $\boldsymbol{d}_j(t)$.

Neighbor Agent, j	**Event time,** t	**State averages,** $\boldsymbol{d}_j(t)$
Agent 1	1	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	2	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	3	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	4	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	5	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
Agent 2	1	$[-0.2360, -0.4365,\ 0.3356,\ 0.0688]^T$
	2	$[-1.5885,\ 0.9261,\ 0.4067,\ 1.6100]^T$
	3	$[-0.2572,\ 1.5680, -0.6218, -0.1870]^T$
	4	$[-0.0696,\ 1.2436, -0.0115, -0.7308]^T$
	5	$[0.1038, -0.2305, -0.5897,\ 1.3774]^T$

Table 4.2. Solution estimates of the neighbor agents of Agent 5, $\boldsymbol{x}_j(t)$ (cont.)

Agent 3	1	$[-0.1835,\ 0.3670, -0.2294,\ 0.0917]^T$
	2	$[0.3645,\ 1.0028,\ 0.7683,\ 1.1386]^T$
	3	$[0.4936,\ 1.3209,\ 1.3492,\ 1.5768]^T$
	4	$[0.4437,\ 1.4501,\ 1.6783,\ 1.7829]^T$
	5	$[0.3174,\ 1.4613,\ 1.8500,\ 1.9146]^T$
Agent 4	1	$[-1.4857,\ -0.7601,\ 4.2906,\ 6.1563]^T$
	2	$[-2.0626,\ -1.9683,\ 2.6744,\ 6.9943]^T$
	3	$[-1.6298,\ -1.0619,\ 3.8870,\ 6.3656]^T$
	4	$[-1.2555,\ -0.2782,\ 4.9353,\ 5.8220]^T$
	5	$[-1.3849,\ -0.5491,\ 4.5728,\ 6.0100]^T$
Agent 6	1	$[-0.6212,\ 0.8742, -0.1243,\ 0.1143]^T$
	2	$[0.2434,\ 1.5145,\ 1.5068,\ 1.8614]^T$
	3	$[0.5758,\ 1.7663,\ 2.1412,\ 2.5052]^T$
	4	$[0.7003,\ 1.8546,\ 2.3842,\ 2.7753]^T$
	5	$[0.7531,\ 1.8825,\ 2.4898,\ 2.9122]^T$

Next, Agent 5 continues with Phase #2. It first computes a basis for the null space of matrix C_j and appends each basis vector to the columns of matrix M_j, as presented in Table 4.3. Thus, Agent 5 discovers the equation sets of its neighboring agents, as shown in Table 4.4.

In Phase #2.1, the agents check the condition for detecting all random-state faulty agents. As can be inferred from Table 4.3, M_2 was computed as a zero matrix which led to the fact stated in Theorem 3.2. Therefore, Agent 2 is detected as a random-state faulty agent and removed from the in-neighbor set of Agent 5.

Table 4.3. Computed M_j matrices

N. Agent, j	M_j	**N. Agent,** j	M_j
Agent 1	$\begin{bmatrix} 0.878 & 0.182 & 0.086 & 0.084 \\ -0.194 & -0.622 & -0.262 & 0.194 \\ 0.215 & -0.016 & -0.881 & -0.378 \\ -0.378 & 0.746 & -0.193 & -0.044 \\ 0.031 & 0.147 & -0.331 & 0.899 \end{bmatrix}$	**Agent 4**	$\begin{bmatrix} -0.3991 \\ -0.3326 \\ 0.5322 \\ 0.0665 \\ 0.6652 \end{bmatrix}$
Agent 2	$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$	**Agent 6**	$\begin{bmatrix} 0.2085 \\ 0.1390 \\ 0.2780 \\ -0.2085 \\ 0.9036 \end{bmatrix}$
Agent 3	$\begin{bmatrix} -0.3455 \\ 0.6911 \\ -0.4319 \\ 0.1728 \\ 0.4319 \end{bmatrix}$		

Table 4.4. Equations of neighbors of Agent 5.

Equations	**Neighbor Agent,** j
no equation available	Agent 1
no equation available	Agent 2
$-4x_1 + 8x_2 - 5x_3 + 2x_4 = 5$	Agent 3
$-6x_1 - 5x_2 + 8x_3 + x_4 = 10$	Agent 4
$-6x_1 - 4x_2 - 8x_3 + 6x_4 = -26$	Agent 6

Agent 5 can also identify all fixed-state faulty agents in its in-neighbor set by executing Phase #2.2. At first, the projection matrices, $\bar{P}_{ker(A_j)}$, are computed for each neighbor j using the computed matrices of M_j. Note here that the condition given in Theorem 3.3 holds for Agent 1. Therefore, Agent 1 is labeled as a fixed-state faulty agent.

In Phase #2.3, Agent 5 computes the difference vector, $\boldsymbol{h}_j$, for each neighbor j and checks the condition claimed in Theorem 3.4. If the given condition holds, Agent 5 labels the neighboring agent as faulty. In this network, Agent 5 identifies Agent 4 and Agent 6 as faulty agents and removes them from its in-neighbor list.

Later, Agent 5 proceeds to execute the discrete-time distributed algorithm (2.10) with its updated neighbor set, which only includes normal Agent 3. The evolution of the solution estimates of Agent 5 is illustrated in Figure 4.4. As can be seen from Figure 4.4, Agent 5 struggles for the first four time steps, which corresponds to fault detection scheme. After the fifth time step, Agent 5 begins to coordinate with other normal agents since the other normal agents could also detect the faulty agents in their neighbor lists until $t = 6$. This coordination between the normal agents can be observed in Figure 4.5. Moreover, the decline in the total error for the normal agents which indicates the asymptotic convergence is illustrated in Figure 4.6.

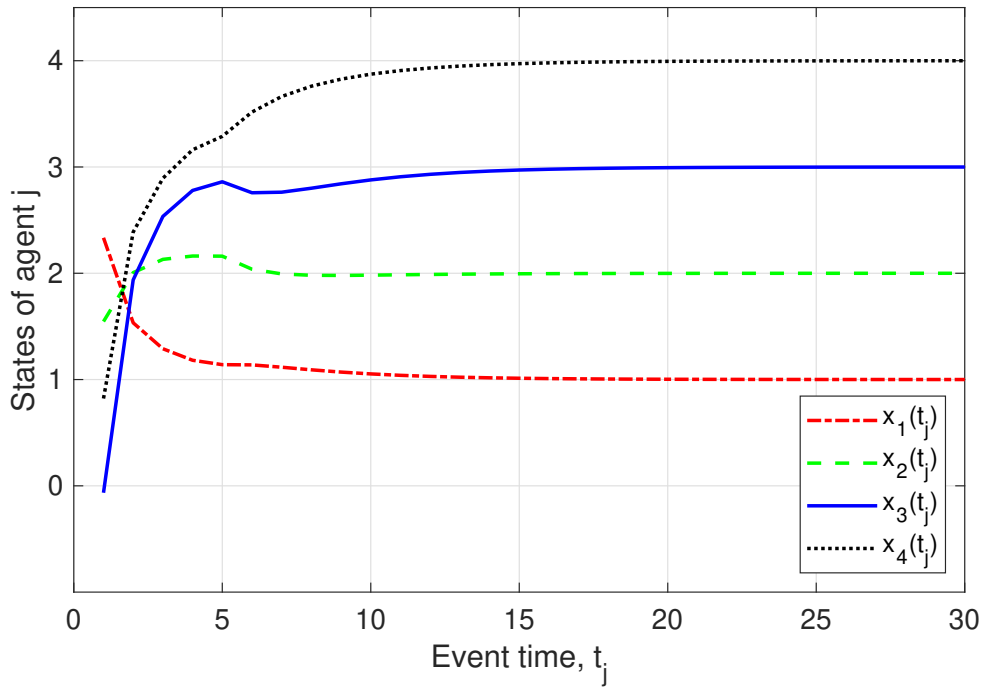


Figure 4.4. Solution estimates of Agent 5

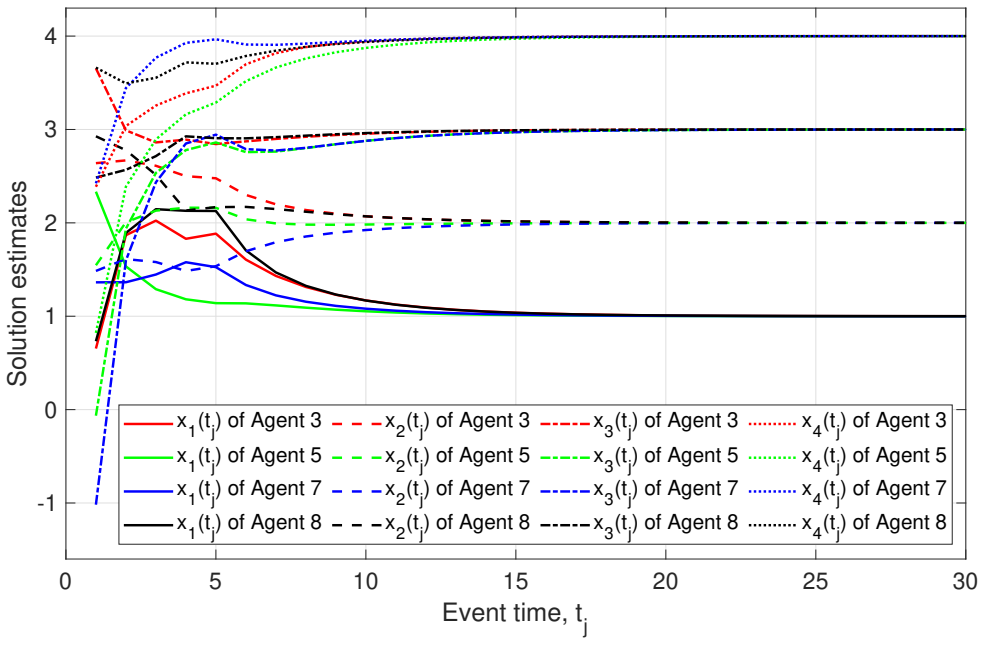


Figure 4.5. Solution estimates of all normal Agent in $\mathcal{G}_f$

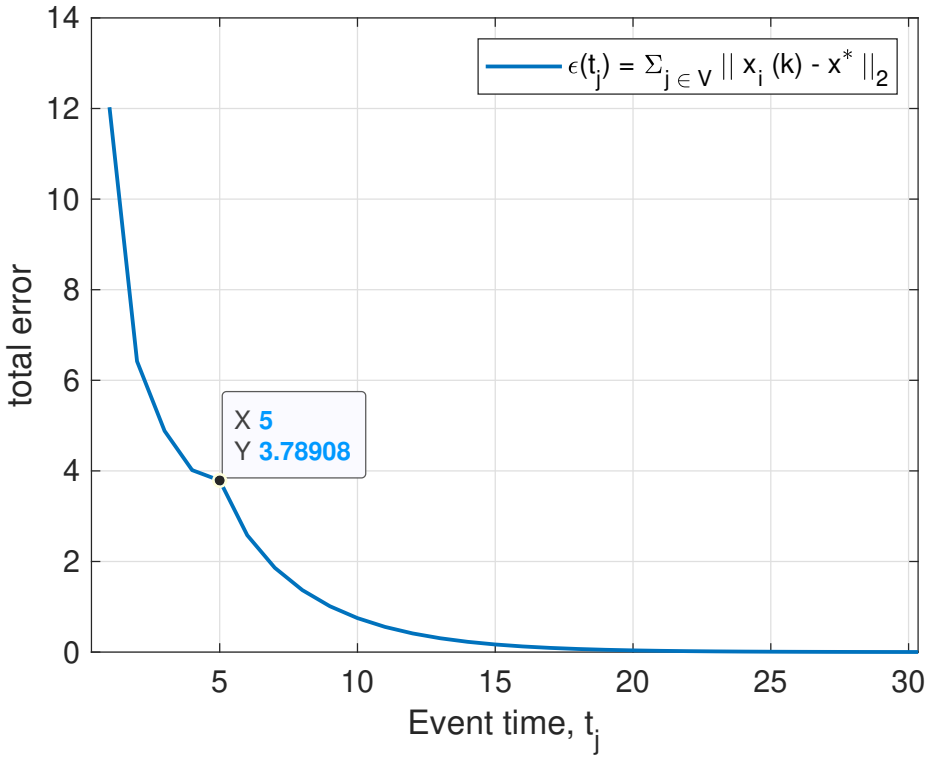


Figure 4.6. Total error of normal agents

4.5. Summary of the Chapter

In this chapter, we have introduced a synchronous fault-tolerant distributed algorithm to solve linear algebraic equations of the form (1.1). Furthermore, we have provided a detailed explanation for the suggested algorithm and analyzed the algorithm in terms of time and space complexities. Lastly, we have illustrated a numerical simulation in which the proposed fault detection algorithm identifies all faulty agents in the neighbor list of each normal agent. Since all agents can label the faulty agents in their neighbor list after an $n - r + 1$ time-steps, we have observed that normal agents achieve asymptotic consensus on a feasible solution to $A\boldsymbol{x} = b$.

5. ASYNCHRONOUS FAULT RESILIENT DISTRIBUTED ALGORITHM

In Chapter 2, we reviewed different types of distributed algorithms to solve linear algebraic equations. However, most of the developed distributed algorithms in the literature are not directly applicable to faulty networks. Therefore, we proposed additional fault detection schemes for averaging consensus-based distributed algorithms to make them resilient to faulty behaviors.

On the other hand, the fault detection algorithm proposed in Chapter 3 cannot be directly applied if each agent updates its state in asynchronous event time sequences. Accordingly, we introduce an asynchronous fault resilient distributed algorithm to detect faulty agents in this chapter. In addition, we provide time and space complexity analysis for the proposed algorithm. Then, we present that normal agents are guaranteed to achieve consensus on a feasible solution to the system of equations of the form (1.1) by using our algorithm.

5.1. The Asynchronous Algorithm

Consider $m > 1$ autonomous agents in the network $\mathcal{G}$ seek to solve the system of the form (1.1) by implementing an averaging-based distributed algorithm such as (2.12). Each agent i estimates a solution for its own equation by utilizing the estimates of each in-neighbor agent j at its event time t_{ik} where $k \geq 1$. However, each agent $j \in \mathcal{N}_i^+$ may transmit its updated solution estimate to agent i in asynchronous event times. Therefore, we assume each agent has its own private event time sequence of $(t_{i1}, t_{i2}, \ldots)$ to update and send information to its neighbors. It should be noted that between two consecutive event times t_{ik} and $t_{i(k+1)}$, the solution estimate of agent i, $\boldsymbol{x}_i(t_{ik})$, is remained constant.

As a result of this private asynchrony among the agents, classical synchronized distributed algorithms cannot be applied for solving linear algebraic equations in the multi-agent network settings. Thus, we consider that each agent in $\mathcal{G}$ utilizes the asynchronous distributed algorithm (2.12) developed in [23] to achieve consensus on a feasible solution to a system of the form $A\boldsymbol{x} = b$.

On the other hand, bounded computation and communication delays are considered in [23]. They proposed that agent i can utilize the received data from its in-neighbor agent j at delayed time sequences, $\delta_{ij}(k), (k = 1, 2, \ldots)$. This time delay definition includes both the *transmission time* and the *hold time*. The transmission time defines a bounded time delay on the communication channel between the agents. The hold time specifies a bounded time delay in which agent i holds received data until its event time t_{ik}.

Although the individual time delay is specific to each agent, all normal agents know the maximum delay time, δ_{max}, for the network. Therefore, it is ensured that each normal agent receives and utilizes updated state vectors from their in-neighbors at least once on each interval $[t_{ik},\, t_{ik} + \delta_{max})$, $(k = 1, 2, \ldots)$.

We assume that each agent $j \in \mathcal{V}$ shares two state vectors with each neighboring agent i:

(i) the solution estimate that solves its own equation set, i.e., $\boldsymbol{x}_j(t_{jk}) \in \mathbb{R}^n$ and

(ii) the average of the solution estimates of its in-neighbor agents, i.e.,

$$\boldsymbol{v}_j(t_{jk}) = \frac{1}{m_j(t_{jk})} \sum_{s \in \mathcal{N}_j^+(t_{jk})} \boldsymbol{x}_s(t_{jk} - \delta_{js}(k)) \in \mathbb{R}^n \tag{5.1}$$

where $m_j(t_{jk}) = |\mathcal{N}_j^+(t_{jk})|$ and $k \geq 1$.

Remark 5.1. *Assume each agent $j \in \mathcal{V}$ publishes its data with neighboring agent $i \in \mathcal{N}_j^-$ with time delays. Then, agent i should receive*

$$(n - r_j + 1)\delta_{max} \tag{5.2}$$

where r_j denotes the number of equations of agent j, n denotes the number of unknowns,

and δ_{max} is the maximum delay time for the network, solution estimates from agent j to guarantee for deducing a linear equation system with the same solution set as the system of agent j.

5.2. Description of the Algorithm

In this section, we present the asynchronous fault-resilient discrete-time distributed algorithm for solving linear algebraic equations in detail. The algorithm is provided in Figures 5.1 and 5.2.

Phase #0: In this phase, agent i initializes its solution estimate with a vector that should solve the equation set of agent i. For the initialization, agent i uses the minimum norm solution of the equation set $[A_i \ b_i]$. Then agent i computes the projection matrix onto the kernel of its equation set with (2.11). In addition, agent i determines the number of its in-neighbor agents and assigns the maximum delay on the network. Lastly, agent i generates an empty 2D array for each in-neighbor j to use in the latter phases. We also set a constant small precision value ϵ_c for the simulations.

Phase #1: This phase is crucial for the fault detection phase. Agent i uses a previously generated 2D array to compile the solution estimates of in-neighbors of agent i. On the other hand, agent i receives data from its in-neighbors asynchronously. Thus, agent i should append only the linearly independent state vectors received from its in-neighbor agent j to rows of matrix C_j.

1: $\boldsymbol{x}_i(t_{i1}) \leftarrow A_i^T(A_iA_i^T)^{-1}b_i$ ▶ Phase #0
2: $P_{A_i} \leftarrow I - A_i^T(A_iA_i^T)^{-1}A_i$
3: $\mu_i \leftarrow |\mathcal{N}_i^+(t_{i1})|$
4: $\epsilon_c \leftarrow$ a small precision value
5: $\delta_{max} \leftarrow$ maximum delay on the network
6: **foreach** $j \in \mathcal{N}_i^+(t_{i1})$ **do**
7: Initialize C_j as an empty 2D array
8: **end for**
9: $k \leftarrow 1$
10: **do** ▶ Phase #1
11: Initialize $\boldsymbol{d}_i(t_{ik})$ as an empty vector in $\mathbb{R}^n$
12: **foreach** $j \in \mathcal{N}_i^+(t_{ik})$ **do**
13: $\boldsymbol{d}_j(t_{ik} - \delta_{ij}(k))$ received from j
14: $\boldsymbol{x}_j(t_{ik} - \delta_{ij}(k))$ received from j
15: $w_{ij} \leftarrow 1/\mu_i$
16: $\boldsymbol{d}_i(t_{ik}) \leftarrow \boldsymbol{d}_i(t_{ik}) + w_{ij}\boldsymbol{x}_j\big(t_{ik} - \delta_{ij}(k)\big)$
17: Append the rows of $[\boldsymbol{x}_j(t_{ik})^T, \ -1]$ to C_j
18: **end for**
19: $\boldsymbol{x}_i(t_{i(k+1)}) \leftarrow \boldsymbol{x}_i(t_{ik}) - P_{A_i}\big(\boldsymbol{x}_i(t_{ik}) - \boldsymbol{d}_i(t_{ik})\big)$
20: $k \leftarrow k + 1$
21: **while** $rank(C_j) < n + 1$ or $k < (n - r_j + 1)\delta_{max} + 1$
22: **foreach** $j \in \mathcal{N}_i^+(t_{ik})$ **do** ▶ Phase #2
23: $M_j \leftarrow$ a basis for the $null(C_j)$ ▷ Phase #2.1
24: **if** $||M_j||_1 \neq 0$ **then**
25: $\mathcal{N}_i^+(t_{ik}) \leftarrow \mathcal{N}_i^+(t_{ik}) \setminus j$
26: **continue**
27: **end if**

Figure 5.1. Asynchronous fault resilient distributed algorithm for solving linear algebraic Equations

$\bar{A}_j \leftarrow M_j[1:n][:]^T$ ▷ Phase #2.2

$\bar{P}_{\bar{A}_j} \leftarrow I - \bar{A}_j{}^T(\bar{A}_j\bar{A}_j{}^T)^{-1}\bar{A}_j$

if $||\bar{P}_{\bar{A}_j}||_1 < n^2\epsilon_c$ **then**

$\mathcal{N}_i^+(t_{ik}) \leftarrow \mathcal{N}_i^+(t_{ik}) \setminus j$

continue

end if

$\boldsymbol{h}_j \leftarrow \boldsymbol{x}_j(t_{j(k-1)}) - \boldsymbol{x}_j(t_{j(k)}) - \bar{P}_{\bar{A}_j}\Big(\boldsymbol{x}_j(t_{j(k-1)}) - \boldsymbol{d}_j(t_{j(k-1)})\Big)$ ▷ Phase #2.3

if $||\boldsymbol{h}_j||_1 > n\epsilon_c$ **then**

$\mathcal{N}_i^+(t_{ik}) \leftarrow \mathcal{N}_i^+(t_{ik}) \setminus j$

continue

end if

end for

$m_i \leftarrow |\mathcal{N}_i^+(t_{ik})|$ ▶ Phase #3

do

Initialize $\boldsymbol{v}_i(t_{ik})$ as an empty vector in $\mathbb{R}^n$

foreach $j \in \mathcal{N}_i^+(t_{ik})$ **do**

$w_{ij} \leftarrow 1/m_i$

$\boldsymbol{d}_i(t_{ik}) \leftarrow \boldsymbol{d}_i(t_{ik}) + w_{ij}\boldsymbol{x}_j\big(t_{ik} - \delta_{ij}(k)\big)$

end for

$\boldsymbol{x}_i(t_{i(k+1)}) \leftarrow \boldsymbol{x}_i(t_{ik}) - P_i\Big(\boldsymbol{x}_i(t_{ik}) - \boldsymbol{d}_i(t_{ik})\Big)$

$k \leftarrow k + 1$

while $||\boldsymbol{x}_i(t_{i(k+1)}) - \boldsymbol{x}_i(t_{ik})|| > \epsilon_c$

Figure 5.2. Asynchronous fault resilient distributed algorithm for solving linear algebraic equations (cont.)

Phase #2: In this phase, agent i identifies all faulty agents in its in-neighbor set. We discuss this phase in three sub-phases.

- In Phase #2.1, agent i computes a basis of the null space of matrix C_j and appends each basis to the columns of matrix M_j. Later on, agent i computes the norm of M_j to identify whether agent j is a random-state faulty agent. If the computed value is not equal to zero as proposed in Theorem 3.2, agent i labels agent j as a random-state faulty agent and deletes agent j from its in-neighbor set $\mathcal{N}_i^+$.
- In Phase #2.2, agent i computes the projection matrix $\bar{P}_{ker(\bar{A}_j)}$ used by agent $j \in \mathcal{N}_i^+$ with the help of the matrix M_j computed in the previous phase. Then, agent i computes the norm of the projection matrix of agent j and checks the condition stated in Theorem 3.3 to label agent j as a fixed-state faulty agent.
- Since we assume that each agent utilizes the asynchronous distributed algorithm introduced in (2.12) to update its solution estimate, agent i checks the condition provided in Theorem 3.4 to identify single-faced and double-faced faulty agents in $\mathcal{N}_i^+$. Therefore, agent i computes the difference vector $\boldsymbol{h}_j$ in Phase #2.3. Then, agent i should confirm the condition stated in Corollary 3.5 with the difference vector for labeling agent j as normal agent. Otherwise, agent i labels agent j as faulty. Then, it removes agent j out of its in-neighbor set $\mathcal{N}_i^+$.

Phase #3: Until this phase, agent i identified all faulty agents in its in-neighbor set and deleted them from its in-neighbor set, i.e., $\mathcal{N}_i^+$. Consequently, agent i can now rely on each agent in its fault-free in-neighbor set. Therefore, agent i can update its solution estimate by way of an asynchronous distributed algorithm (2.12) until it is made sure consensus is reached among the normal agents to solve $A\boldsymbol{x} = b$.

5.3. Algorithm Complexity Analysis

In this section, we investigate the time and space complexities of the proposed asynchronous fault resilient discrete-time distributed algorithm illustrated in Figures 5.1

and 5.2. First, we analyze the algorithm's time complexity for each phase individually.

In Phase #0, the initialization procedure includes matrix inversion which we maintain the classical QR-decomposition method described in [42] for this operation. Thus, this computation takes $\mathcal{O}(r_j^2 n)$ times. On the other hand, the projection matrix computation takes $\mathcal{O}(r_j n^2)$ times. Accordingly, the total time cost is $\mathcal{O}(r_j n^2)$ for Phase #0.

In Phase #1, the condition for the "while" loop indicates that computation may occur $((n - r_j + 1)\delta_{max})$ times. We observe this amount of iteration as the worst case when analyzing the total time complexity. Thus, we find the total time cost as $\mathcal{O}(n^3 \delta_{max} \mu_i)$ in the worst-case scenario.

In Phase #2.1, the singular value decomposition (SVD) method described in [43] is used to compute a basis for the null space of matrix C_j of size $((n - r_j + 1)\delta_{max} \times n + 1)$. This process takes $\mathcal{O}(n^3 \delta_{max})$ times. Moreover, the computation of 1-norm of an $(n + 1 \times r_j)$ matrix takes $\mathcal{O}(r_j n)$ times [44]. Later, the projection matrix is found with the time cost of $\mathcal{O}(r_j n^2)$, and computation of the norm of this projection matrix takes $\mathcal{O}(n^2)$ in Phase #2.2. Phase #2.3 takes $\mathcal{O}(n^2)$ times due to the matrix and vector multiplications in the difference vector computation. Also, the computation of the norm of an $n \times 1$ vector requires $\mathcal{O}(n)$. All computations are repeated μ_i times because of the "foreach" loop. Thus, the total cost of Phase #2 is found as $\mathcal{O}(n^3 \delta_{max} \mu_i)$ in the worst-case scenario of the maximum delay δ_{max} being reached for each agent.

Finally, the matrix and vector multiplications take $\mathcal{O}(n^2)$ time in Phase #3. All in all, the total time cost for the asynchronous fault resilient discrete-time distributed algorithm is found as $\mathcal{O}(n^3 \delta_{max} \mu_i)$ in the worst case for bounded δ_{max}.

We now examine the space complexities of each phase of the proposed asynchronous fault resilient distributed algorithm. In Phase #0, the 2D matrix initialization requires $\mathcal{O}(n^2 \mu_i)$ space as the maximum space among the other vector and matrix

initializations. In Phase #1, each assignment of the vector of size $n \times 1$ in the "foreach" loop requires $\mathcal{O}((n - r_j + 1)\delta_{max}\mu_i)$ space in the worst-case scenario since δ_{max} is the upper bound for the condition of the "while" loop. Thus, the total space required by Phase #1 is $\mathcal{O}(n^2\delta_{max}\mu_i)$. Finally, in Phase #2, there are vector and 2D matrix assignments, and the total space requirement is found as $\mathcal{O}(n^2\mu_i)$. In total, the space complexity of the algorithm is $\mathcal{O}(n^2\delta_{max}\mu_i)$.

5.4. A Numerical Example

As for the numerical example, we reconsider the network $\mathcal{G}_f$ illustrated in Figure 4.3. There are eight agents in the network who are unfamiliar with the identities of their neighboring agents at the beginning of the process.

In this example, the system provided in (4.2) has a unique solution of $[1, 2, 3, 4]^T$. However, none of the agents is aware of this unique solution to the system. Thus, normal agents try to achieve this unique solution by interchanging their data through the transmission channels illustrated in Figure 4.3. Nevertheless, the event time cycle of each agent may differ from the other agents since we consider the asynchrony of the event times. On the other hand, we assume that the maximum delay time for this network δ_{max} is bounded and known by each normal agent $i \in \mathcal{V}_n$. It should also be noted that each agent i shares two separate vectors: $\boldsymbol{x}_i(t_{ik})$ and $\boldsymbol{d}_i(t_{ik}) \in \mathbb{R}^n$ with its out-neighbors at its event times.

We reconsider Agent 5 which is one of the normal agents in $\mathcal{G}_f$ illustrated in Figure 4.3. Agent 5 knows only its own equation set given in (4.3). We monitor the evolution of its solution estimate to achieve consensus with other normal agents. However, Agent 5 should ensure that each in-neighbor agent, $\mathcal{N}_5^+$, is a normal agent before applying an asynchronous discrete-time distributed algorithm such as algorithm (2.12) to achieve asymptotic consensus. Therefore, Agent 5 should identify the true intentions of all its in-neighbor agents using fault detection algorithm illustrated in Figures 5.1 and 5.2.

In Phase #0 of the algorithm illustrated in Figure 5.1, Agent 5 initializes its solution estimate as a feasible solution to its equation set provided in (4.3). Moreover, Agent 5 computes its projection matrix by using its equation set in line with (2.11). However, Agent 5 determines the number of in-neighbor agents at the beginning of the process and initializes a 2D matrix called C_j for each agent $j \in \mathcal{V}_5^+$. Finally, the maximum delay time is set to three, i.e. $\delta_{max} = 3$.

Next, Agent 5 listens to its in-neighbors at its event times for their data and compiles them into the previously generated 2D matrix C_j in Phase #1. As mentioned before, each in-neighbor of Agent 5 shares two separate vectors with Agent 5. The solution estimates of each in-neighbor agent of Agent 5 are given in Table 5.1. In addition, the state vectors $\boldsymbol{d}_j$ published from the in-neighbors of Agent 5 are provided in Table 5.2.

Table 5.1. Solution estimates of the neighbor agents of Agent 5, $\boldsymbol{x}_j(t_{5k})$.

Neighbor Agent, j	**Event time,** t_{5k}	**Estimated Solution,** $\boldsymbol{x}_j(t_{5k})$
Agent 1	1	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	2	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	3	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	4	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
	5	$[1.7708,\ 2.8492, -0.7640,\ 2.1233]^T$
Agent 2	1	$[0.1825,\ 1.5651, -0.0845,\ 1.6039]^T$
	2	$[0.1825,\ 1.5651, -0.0845,\ 1.6039]^T$
	3	$[0.0983,\ 0.0414, -0.7342, -0.0308]^T$
	4	$[0.0983,\ 0.0414, -0.7342, -0.0308]^T$
	5	$[0.2323,\ 0.4264, -0.3728, -0.2365]^T$
Agent 3	1	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	2	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	3	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	4	$[1.6989,\ 2.6894,\ 3.1007,\ 2.8919]^T$
	5	$[1.6989,\ 2.6894,\ 3.1007,\ 2.8919]^T$

Table 5.1. Solution estimates of the neighbor agents of Agent 5, $\boldsymbol{x}_j(t_{5k})$. (cont.)

Agent 4	1	$[-0.4762, -0.3968,\ 0.6349,\ 0.0794]^T$
	2	$[-0.4762, -0.3968,\ 0.6349,\ 0.0794]^T$
	3	$[0.1010,\ 0.1853,\ 1.4382,\ 0.0266]^T$
	4	$[0.1010,\ 0.1853,\ 1.4382,\ 0.0266]^T$
	5	$[0.3732,\ 0.3050,\ 1.7203,\ 0.0018]^T$
Agent 6	1	$[1.0263,\ 0.6842,\ 1.3684, -1.0263]^T$
	2	$[1.0263,\ 0.6842,\ 1.3684, -1.0263]^T$
	3	$[1.0263,\ 0.6842,\ 1.3684, -1.0263]^T$
	4	$[1.2983,\ 1.0941,\ 1.2658, -0.6179]^T$
	5	$[1.2983,\ 1.0941,\ 1.2658, -0.6179]^T$

Table 5.2. Average of received states from neighbor of Agent 5, $\boldsymbol{d}_j(k)$.

Neighbor Agent, j	**Event time,** t_{5k}	**Average states,** $\boldsymbol{d}_j(k)$
Agent 1	1	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	2	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	3	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	4	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
	5	$[3.5784,\ 0.7694, -1.3499,\ 3.0349]^T$
Agent 2	1	$[0.8731,\ 0.1853, -2.4878,\ 0.1706]^T$
	2	$[0.8731,\ 0.1853, -2.4878,\ 0.1706]^T$
	3	$[-0.5254,\ 0.3493,\ 0.1980, -0.5998]^T$
	4	$[-0.5254,\ 0.3493,\ 0.1980, -0.5998]^T$
	5	$[1.4109,\ 1.0889, -0.7958,\ 1.2149]^T$
Agent 3	1	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	2	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	3	$[0.6539,\ 2.6391,\ 3.6541,\ 2.3865]^T$
	4	$[1.5377,\ 2.4908,\ 3.0490,\ 3.2348]^T$
	5	$[1.5377,\ 2.4908,\ 3.0490,\ 3.2348]^T$

Table 5.2. Average of received states from neighbor of Agent 5, $\boldsymbol{d}_j(k)$. (cont.)

Agent 4	1	$[-1.4857,\ -0.7601,\ 4.2906,\ 6.1563]^T$
	2	$[-1.4857,\ -0.7601,\ 4.2906,\ 6.1563]^T$
	3	$[-2.0626,\ -1.9683,\ 2.6744,\ 6.9943]^T$
	4	$[-2.0626,\ -1.9683,\ 2.6744,\ 6.9943]^T$
	5	$[-1.6298,\ -1.0619,\ 3.8870,\ 6.3656]^T$
Agent 6	1	$[-0.6212,\ 0.8742, -0.1243,\ 0.1143]^T$
	2	$[0.2434,\ 1.5145,\ 1.5068,\ 1.8614]^T$
	3	$[0.5758,\ 1.7663,\ 2.1412,\ 2.5052]^T$
	4	$[0.7003,\ 1.8546,\ 2.3842,\ 2.7753]^T$
	5	$[0.7531,\ 1.8825,\ 2.4898,\ 2.9122]^T$

In Phase #2, Agent 5 executes the faulty detection algorithm. First, it computes a basis for the null space of matrix C_j for each of its in-neighbor agent j. Then it appends the computed basis to the columns of the matrix M_j. The generated M_j matrices are presented in Table 5.3.

In Phase #2.1, Agent 5 checks the condition for identifying the random-state faulty agent. As can be observed from Table 5.3, the equation set of Agent 2 was determined as the zero set by Agent 5. Thus, Agent 2 was identified as a random-state faulty agent. Then Agent 5 broke ties with Agent 2.

In the next phase, Agent 5 determined the fixed-state faulty agents in its in-neighbor set by checking the condition defined in Theorem 3.3. Agent 5 identifies Agent 1 as faulty, since the computed projection matrix of Agent 1 did not satisfy Theorem 3.3. Therefore, Agent 1 is also deleted from the in-neighbor list of Agent 5.

Lastly, Agent 5 made use of Theorem 3.4 in order to identify the single-faced and double-faced faulty agents in Phase #2.3. For each of its remaining in-neighbors, the difference vector $\boldsymbol{h}_j$ was computed by Agent 5. Then, Agent 4 and Agent 6 are detected as faulty agents and removed by Agent 5 from its in-neighbor list.

Table 5.3. Computed M_j matrices

N. Agent, j	M_j	**N. Agent,** j	M_j
Agent 1	$\begin{bmatrix} 0.878 & 0.182 & 0.086 & 0.084 \\ -0.194 & -0.622 & -0.262 & 0.194 \\ 0.215 & -0.016 & -0.881 & -0.378 \\ -0.378 & 0.746 & -0.193 & -0.044 \\ 0.031 & 0.147 & -0.331 & 0.899 \end{bmatrix}$	**Agent 4**	$\begin{bmatrix} -0.3991 \\ -0.3326 \\ 0.5322 \\ 0.0665 \\ 0.6652 \end{bmatrix}$
Agent 2	$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$	**Agent 6**	$\begin{bmatrix} 0.2085 \\ 0.1390 \\ 0.2780 \\ -0.2085 \\ 0.9036 \end{bmatrix}$
Agent 3	$\begin{bmatrix} -0.3455 \\ 0.6911 \\ -0.4319 \\ 0.1728 \\ 0.4319 \end{bmatrix}$		

Table 5.4. Equations of neighbors of Agent 5.

Equations	**N. Agent,** j
no equation available	1
no equation available	2
$-4x_1 + 8x_2 - 5x_3 + 2x_4 = 5$	3
$-6x_1 - 5x_2 + 8x_3 + x_4 = 10$	4
$-6x_1 - 4x_2 - 8x_3 + 6x_4 = -26$	6

Up to now, Agent 5 has detected all faulty agents in its in-neighbor list and removed them from this list. In Phase #3, Agent 5 executes the asynchronous discrete-time distributed algorithm (2.12) with its updated in-neighbor list until a consensus is achieved with the other normal agents. The evolution of the solution estimates of Agent 5 is illustrated in Figure 5.3. Moreover, the consensus on the unique solution of $[1, 2, 3, 4]^T$ among the normal agents can be observed in Figures 5.4 and 5.5.

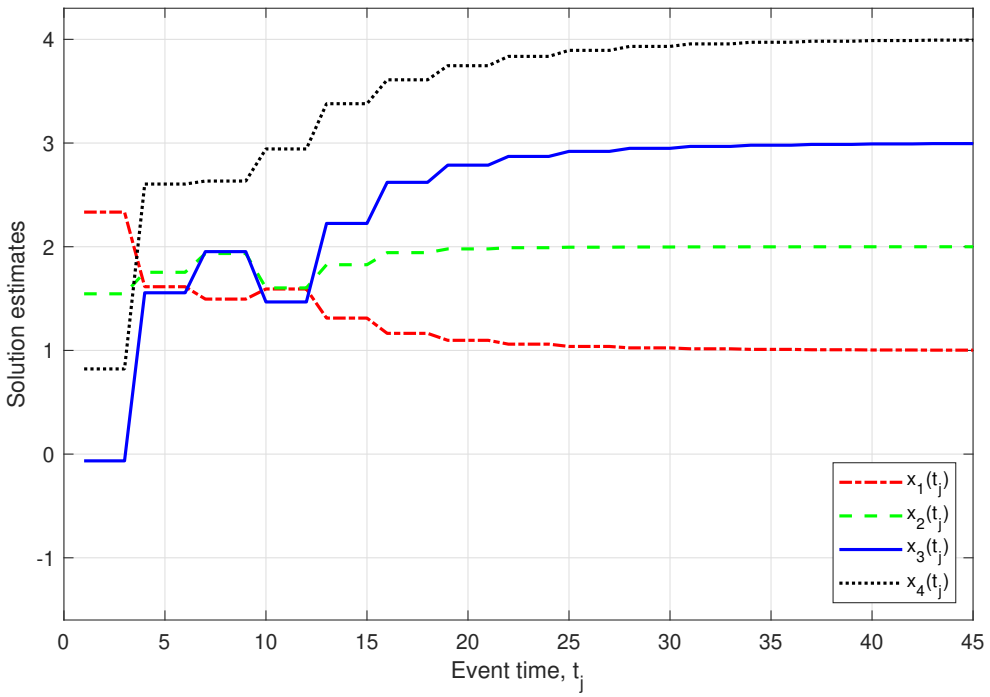


Figure 5.3. Solution estimates of Agent 5

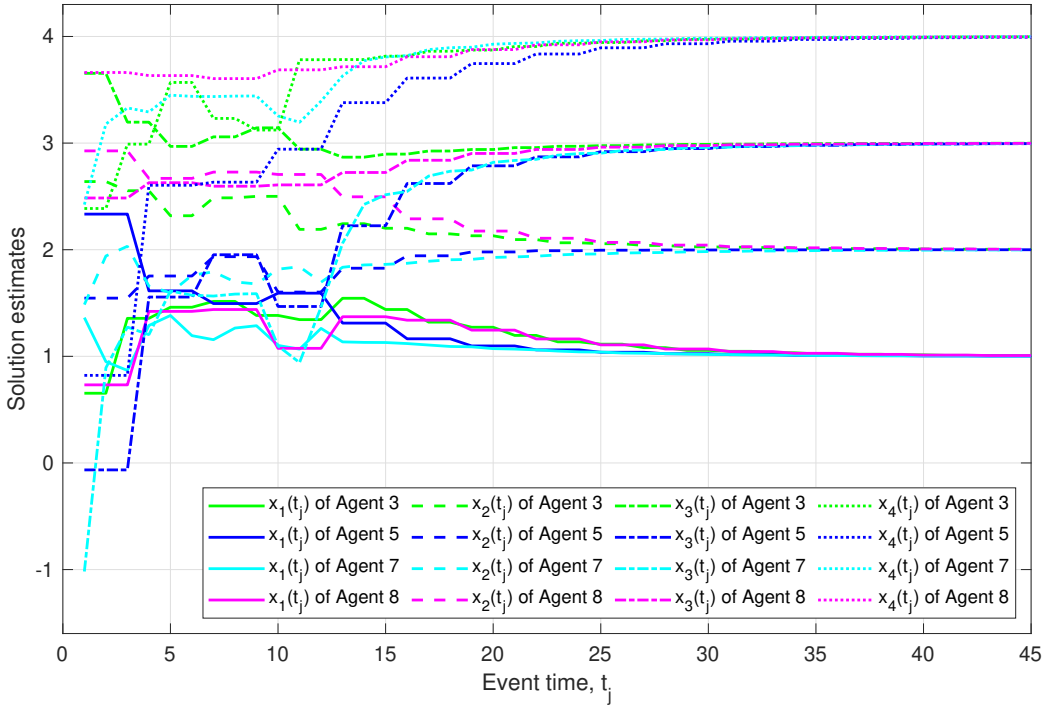


Figure 5.4. Solution estimates of all normal Agents in $\mathcal{G}_f$

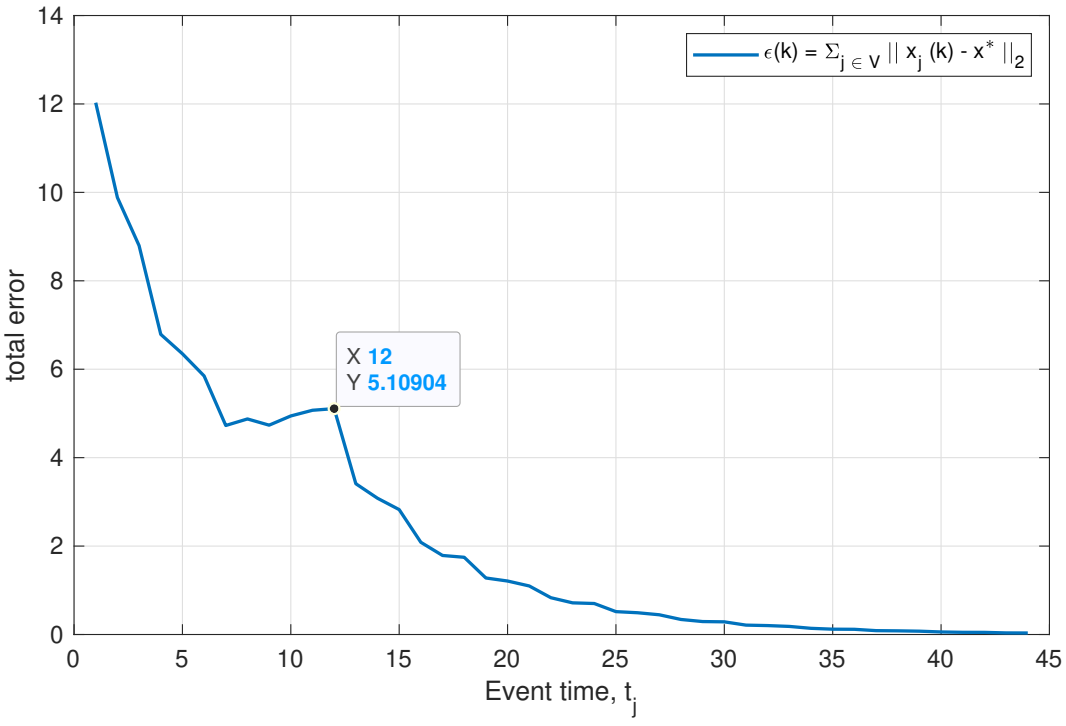


Figure 5.5. Total estimation error of the normal agents

5.5. Summary of the Chapter

In this chapter, we have proposed a fault-resilient asynchronous distributed algorithm to solve linear algebraic equations. We have split the process into four phases, including the initialization phase. We have guaranteed that each normal agent identifies all faulty agents in its neighbor set with the asynchronous fault detection phase. Thus, normal agents achieve consensus in the presence of faulty agents even if they do not have a common event time sequences. We have also implemented complexity analysis for the suggested algorithm. Lastly, we have provided an example to illustrate the feasibility of the suggested algorithm.

6. CONCLUSION

In this thesis, we have studied fault resiliency in distributed algorithms to solve linear algebraic equations in multi-agent networks. In multi-agent networks, we consider that multiple autonomous agents endeavor to achieve consensus distributively on the same feasible solution for a system of equations of the form (1.1). Since privacy and security concerns arise as the number of agents increases, we discuss the effect of faulty participants in the network throughout the thesis.

In most studies on distributed algorithms for solving linear equations, the equation system of neighboring agents are assumed private due to security and privacy concerns [7, 22, 26]. However, a method developed in [45] reveals that this assumption is not restrictive for agents to determine the equation sets of their neighbors. On the other hand, the sufficiency condition proposed in [45] has been corrected to be $n-r+1$ when $r \geq 1$ in Chapter 3 since the complete solution set of a non-homogeneous system $A\boldsymbol{x} = b$ with $rank(A)$ and $rank([A\ b]) = r$, has at most $n - r + 1$ linearly independent solutions.

In Chapter 3, we have introduced four different fault models: *random-state*, *fixed-state*, *single-faced*, and *double-faced*. Characteristics of these fault models originated from the well-known fault definitions of Symmetric and Asymmetric (Byzantine) faults described in [39]. Moreover, we have studied the theoretical basis for identifying these faulty agents in line with their characteristics. On the other hand, we have also discussed the applicability of fault detection schemes in continuous-time systems. The significance of the proposed fault detection procedures is that we do not require prior knowledge of faulty agents in the network.

In distributed systems, agents communicate with each other via communication channels to achieve consensus. However, data transmission between agents takes time, depending on the distance between agents. Besides, agents might not share a com-

mon event time sequence to update their states. Therefore, any delay in the data transmission may influence the overall results adversely, and the faulty parties may be impossible to detect in delayed networks. Thus, we have also studied the synchronization of the agents' event times.

In Chapter 4, we have proposed a fault-resilient distributed algorithm to solve the linear equations while all the individual agents perform their tasks in perfect time synchronization with the other agents. In addition, we have analyzed the time and space complexities of the developed algorithm. Also, we have supported our findings with numerical simulations.

In Chapter 5, we have developed an asynchronous fault resilient distributed algorithm to deal with the adverse effects of communication delays. Moreover, the algorithm complexity analysis has been presented for the proposed algorithm. It was shown that the time and space complexities of the suggested algorithm are related to δ_{max}.

In future studies, it is planned to develop fault detection algorithms of more sophisticated fault models than those proposed here. In addition, further analyses of the continuous-time fault detection algorithms will be performed. Moreover, we will study to improve the time and space complexities of the proposed algorithms. Lastly, it is planned to examine the extensions on fault detection schemes for cases where agents utilize communication-efficient distributed algorithms.

REFERENCES

1. Hackbusch, W., *Iterative Solution of Large Sparse Systems of Equations*, Springer, New York, 1994.

2. Horn, R. A. and C. R. Johnson, *Matrix Analysis*, Cambridge University Press, New York, 2012.

3. Young, D., "Iterative Methods for Solving Partial Difference Equations of Elliptic Type", *Transactions of the American Mathematical Society*, Vol. 76, No. 1, pp. 92–111, 1954.

4. Koç, C. K., A. Güvenç and B. Bakkaloğlu, "Exact Solution of Linear Equations on Distributed-memory Multiprocessors", *Parallel Algorithms and Applications*, Vol. 3, No. 1-2, pp. 135–143, 1994.

5. Andersson, C., "Solving Linear Equations on Parallel Distributed Memory Architectures by Extrapolation", *Technical Report, Royal Institute of Technology*, Vol. 32, No. 4, pp. 156–164, 1997.

6. Usui, M., H. Niki and T. Kohno, "Adaptive Gauss-Seidel Method for Linear Systems", *International Journal of Computer Mathematics*, Vol. 51, No. 1-2, pp. 119–125, 1994.

7. Wang, P., S. Mou, J. Lian and W. Ren, "Solving a System of Linear Equations: From Centralized to Distributed Algorithms", *Annual Reviews in Control*, Vol. 47, No. 1, pp. 306–322, 2019.

8. You, K., S. Song and R. Tempo, "A Networked Parallel Algorithm for Solving Linear Algebraic Equations", *2016 IEEE 55th Conference on Decision and Control (CDC)*, Las Vegas, USA, 2016.

9. Anderson, B., S. Mou, A. S. Morse and U. Helmke, "Decentralized Gradient Algorithm for Solution of a Linear Equation", ArXiv:1509.04538 [cs], 2015.

10. Lynch, N. A., *Distributed Algorithms*, Elsevier, San Francisco, USA, 1996.

11. Gazi, V., B. Fidan, L. Marques, R. Ordonez, E. Kececi and M. Ceccarelli, "Robot Swarms: Dynamics and Control", *Mobile Robots for Dynamic Environments*, pp. 79–107, ASME Press, New York, USA, 2015.

12. Yang, T., X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin and K. H. Johansson, "A survey of Distributed Optimization", *Annual Reviews in Control*, Vol. 47, No. 1, pp. 278–305, 2019.

13. Hespanha, J. P., P. Naghshtabrizi and Y. Xu, "A Survey of Recent Results in Networked Control Systems", *Proceedings of the IEEE*, Vol. 95, No. 1, pp. 138–162, 2007.

14. Bidram, A., F. L. Lewis and A. Davoudi, "Distributed Control Systems for Small-Scale Power Networks: Using Multiagent Cooperative Control Theory", *IEEE Control Systems*, Vol. 34, No. 6, pp. 56–77, 2014.

15. Nedic, A. and J. Liu, "Distributed Optimization for Control", *Annual Review of Control, Robotics, and Autonomous Systems*, Vol. 1, No. 1, pp. 77–103, 2018.

16. Anderson, J. D. and J. Wendt, *Computational Fluid Dynamics*, Springer, Berlin, 1995.

17. Frank, A., D. Fabregat-Traver and P. Bientinesi, "Large-scale Linear Regression: Development of High-performance Routines", *Applied Mathematics and Computation*, Vol. 275, No. 1, pp. 411–421, 2016.

18. Silvestre, D., J. Hespanha and C. Silvestre, "A PageRank Algorithm Based on Asynchronous Gauss-Seidel Iterations", *2018 Annual American Control Conference*

(ACC), Milwaukee, WI, USA, 2018.

19. Cihan, O., "Distributed Solution of Road Lighting Problem Over Multi-Agent Networks", *Sakarya University Journal of Computer and Information Sciences*, Vol. 3, No. 2, pp. 88–97, 2020.

20. Ciftci, O. and O. Cihan, "Solution of a Traffic Network Problem by Using A Continuous-Time Distributed Algorithm", *21st National Conference on Automatic Control*, Muğla, Turkey, 2019.

21. Pasqualetti, F., R. Carli and F. Bullo, "Distributed Estimation via Iterative Projections with Application to Power Network Monitoring", *Automatica*, Vol. 48, No. 5, pp. 747–758, 2012.

22. Mou, S., J. Liu and A. S. Morse, "A Distributed Algorithm for Solving a Linear Algebraic Equation", *IEEE Transactions on Automatic Control*, Vol. 60, No. 11, pp. 2863–2878, 2015.

23. Mou, S. and B. D. O. Anderson, "Eigenvalue Invariance of Inhomogeneous Matrix Products in Distributed Algorithms", *IEEE Control Systems Letters*, Vol. 1, No. 1, pp. 11–19, 2017.

24. Liu, J., S. Mou and A. S. Morse, "Asynchronous Distributed Algorithms for Solving Linear Algebraic Equations", *IEEE Transactions on Automatic Control*, Vol. 63, No. 2, pp. 372–385, 2018.

25. Wang, X., S. Mou and D. Sun, "Improvement of a Distributed Algorithm for Solving Linear Equations", *IEEE Transactions on Industrial Electronics*, Vol. 64, No. 4, pp. 3113–3117, 2017.

26. Yang, M. and C. Y. Tang, "A distributed Algorithm for Solving General Linear Equations Over Networks", *2015 54th IEEE Conference on Decision and Control (CDC)*, Osaka, Japan, 2015.

27. LeBlanc, H. J., H. Zhang, X. Koutsoukos and S. Sundaram, "Resilient Asymptotic Consensus in Robust Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 31, No. 4, pp. 766–781, 2013.

28. Haseltalab, A. and M. Akar, "Approximate Byzantine Consensus in Faulty Asynchronous Networks", *2015 American Control Conference (ACC)*, Chicago, USA, 2015.

29. Oksuz, H. Y. and M. Akar, "Distributed Consensus in Resilient Networks", *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, Melbourne, Australia, 2017.

30. Zhang, H. and S. Sundaram, "Robustness of Information Diffusion Algorithms to Locally Bounded Adversaries", *2012 American Control Conference (ACC)*, Montreal, Canada, 2012.

31. Cao, H.-T., T. E. Gibson, S. Mou and Y.-Y. Liu, "Impacts of Network Topology on the Performance of a Distributed Algorithm Solving Linear Equations", *2016 IEEE 55th Conference on Decision and Control (CDC)*, Las Vegas, USA, 2016.

32. Liu, J., A. S. Morse, A. Nedic and T. Başar, "Exponential Convergence of a Distributed Algorithm for Solving Linear Algebraic Equations", *Automatica*, Vol. 83, No. 1, pp. 37–46, 2017.

33. Anderson, B., S. Mou, A. Morse and U. Helmke, "Decentralized Gradient Algorithm for Solution of a Linear Equation", *Numerical Algebra, Control and Optimization*, Vol. 6, No. 3, pp. 319–328, 2016.

34. Liu, J., X. Chen, T. Başar and A. Nedić, "A Continuous-Time Distributed Algorithm for Solving Linear Equations", *2016 American Control Conference (ACC)*, Boston, MA, USA, 2016.

35. Wang, X., S. Mou and S. Sundaram, "A Resilient Convex Combination for

Consensus-based Distributed Algorithms", ArXiv:1806.10271 [cs], 2018.

36. Tverberg, H., "A Generalization of Radon's Theorem", *Journal of the London Mathematical Society*, Vol. 1, No. 1, pp. 123–128, 1966.

37. Dolev, D., N. A. Lynch, S. S. Pinter, E. W. Stark and W. E. Weihl, "Reaching Approximate Agreement in the Presence of Faults", *Journal of the ACM (JACM)*, Vol. 33, No. 3, pp. 499–516, 1986.

38. Lamport, L. and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", *Journal of the ACM (JACM)*, Vol. 32, No. 1, pp. 52–78, 1985.

39. Krings, A. W. and Z. Ma, "Fault-models in Wireless Communication: Towards Survivable Ad Hoc Networks", *MILCOM 2006-2006 IEEE Military Communications Conference*, Washington, USA, 2006.

40. Azadmanesh, M. H. and R. M. Kieckhafer, "Exploiting Omissive Faults in Synchronous Approximate Agreement", *IEEE Transactions on Computers*, Vol. 49, No. 10, pp. 1031–1042, 2000.

41. Strang, G., *Linear Algebra and its Applications*, Thomson, Brooks/Cole, Belmont, CA, USA, 2006.

42. Tygert, M., "A Fast Algorithm for Computing Minimal-Norm Solutions to Underdetermined Systems of Linear Equations", ArXiv:0905.4745 [cs], 2009.

43. Vasudevan, V. and M. Ramakrishna, "A Hierarchical Singular Value Decomposition Algorithm for Low Rank Matrices", ArXiv:1710.02812 [cs], 2017.

44. Lewis, A. D., "A Top Nine List: Most Popular Induced Matrix Norms", 2010, `https://mast.queensu.ca/~andrew/notes/pdf/2010a.pdf`, accessed in June 2022.

45. Cihan, O., "Rapid Solution of Linear Equations with Distributed Algorithms over Networks", *IFAC-PapersOnLine*, Vol. 52, No. 25, pp. 467–471, 2019.