ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2012, Vol. 41, No. 2

A Simple and Efficient Signature-Based Consensus Protocol in the Asynchronous Distributed System

Chien-Fu Cheng[1, 2, *], Kuo-Tang Tsai[2], Hsien-Chun Liao[1]

[1] Department of Computer Science and Information Engineering,
[2] Graduate Institute of Networking and Communication,
Tamkang University
No.151, Yingzhuan Rd., Tamsui Dist., New Taipei City 251, Taiwan (R.O.C.)
e-mail: cfcheng@mail.tku.edu.tw

crossref http://dx.doi.org/10.5755/j01.itc.41.2.931

Abstract. The consensus problem in distributed systems is mainly solved by message exchange. Most of previous consensus algorithms rely on exchange of oral messages to achieve consensus among processors. As oral messages are susceptible to influences from malicious attackers, this type of consensus protocols usually requires a large number of rounds of message exchange, and the complexity of message exchange is also excessively high. In light of this drawback of oral message-based consensus algorithms, some scholars proposed signed message-based consensus algorithm to reduce the number of rounds of message exchange required. However, some signed message-based consensus algorithms still have certain drawbacks which make them ineffective in some conditions. To address this issue, we propose a new signed message-based consensus algorithm in this paper. We integrate the concept of grouping into the proposed algorithm and find the best number of groups through mathematical analysis to further reduce the rounds of message exchange required. In other words, the proposed algorithm makes use of digital signature and the concept of grouping to solve the consensus problem. This algorithm can not only increase the fault-tolerance of distributed systems but also significantly reduce the rounds of message exchange required to achieve consensus.

Keywords: distributed consensus problem; oral message; signed message; dormant fault; malicious fault and grouping.

1. Introduction

With high scalability, distributed systems are less likely to have single point of failure (SPOF) problems. This advantage has considerably increased the importance and applications of distributed systems. Because a distributed system consists of a number of processors that work together in a network to provide computing power, how to maintain normal operation of the system in the presence of faulty components is an important issue. This issue is also called the fault-tolerance problem in distributed systems. The consensus problem is a well-known type of fault-tolerance problems [7][19]. By solving the consensus problem, we can enhance the reliability and fault-tolerance of distributed systems. An introduction of the consensus problem is provided as follows.

1.1. The Consensus problem

In any distributed system, some processors may operate abnormally due to damage of an internal component, external disturbance or malicious attacks. The abnormal operation of these processors may affect the computing results of the entire system. Helping distributed systems keep away disturbance of faulty components and accurately perform assigned tasks is the primary goal of consensus algorithms. The consensus problem is usually solved by designating one or multiple commanders to broadcast the intended initial value and performing message exchange among processors to make all processors ultimately agree upon the consensus value.

There are two types of the consensus problem: (1) The consensus problem with a single commander: In a distributed system consisting of *n* processors, one processor is assigned to be the commander with an initial value. This problem is also called The Byzantine Agreement Problem [3][16][17] . (2) The consensus problem with *n* commanders: In a distributed system consisting of *n* processors, all processors are commanders respectively having an initial value [4][5][21]. Although these two types of the consensus problems differ in the number of commanders, the second type of the problem can be solved using the consensus protocol for the first type of the problem. The consensus problem with *n*

[*] Corresponding author

commanders can be solved by running *n* copies of the protocol for the problem with a single commander in parallel. Through *n* runs of the protocol, each processor in the problem with multiple commanders will get *n* agreement values. Later, we can select the majority value to achieve the consensus. Protocols designed to deal with the consensus problem with a single commander should satisfy the following conditions:

(Termination): ***All*** non-faulty processors agree on the same value;

(Validity$_{sv}$): If the commander is non-faulty, then ***all*** non-faulty processors agree on the initial value that the commander sends.

Protocols designed to deal with the consensus problem with n commanders should satisfy the following conditions:

(Termination): ***All*** non-faulty processors agree on the same value;

(Validity$_{mv}$): If the initial value of each non-faulty processor is v_i, then ***all*** non-faulty processors agree on v_i.

We know that solution to the consensus problem relies on message exchange. Past research has classified messages exchanged by use of the digital signature technology into oral messages and signed messages [17]. The algorithms that achieve consensus by exchanging oral messages are called Oral Message-based Consensus Protocol (OMC protocol), and those that achieve consensus by exchanging signed messages are called Signed Message-based Consensus Protocol (SMC protocol). The use of digital signature is a key factor affecting the fault-tolerance of a consensus protocol. Given n processors in a distributed computing network, OMC protocols can tolerate $\lfloor(n-1)/3\rfloor$ faulty processors [17]. In the same setting, SMC protocols can tolerate more than $\lfloor(n-1)/3\rfloor$ faulty processors due to the use of digital signature. For example, the SM algorithm proposed by Lamport et al. [17] and Quick Consensus algorithm proposed by Dalui et al. [8] can tolerate n-2 faulty processors. We will give a brief introduction of the above protocols in Section 1.2.

1.2. An Overview

OM algorithm is the first OMC protocol. It was introduced in a paper co-authored by Lamport, Shostak, and Pease in 1982 [17]. In this paper, the authors also developed an SMC protocol called SM algorithm [17]. OM algorithm can tolerate $\lfloor(n-1)/3\rfloor$ faulty processors whereas SM can tolerate n-2 faulty processors. Both algorithms require $\lfloor(n-1)/3\rfloor+1$ rounds of message exchange, and each round of message involves three steps as follows (1) sends messages to other processors, (2) receives messages from other processors, and (3) local processing [3][12][17]. Due to high complexity of message exchange, many later researchers have proposed solutions to reduce the complexity.

For OMC protocols, some researchers proposed to determine termination of message exchange based on sufficiency of messages collected. This kind of algorithm is called the Eventual Consensus Algorithm (ECA). Another algorithm which requires a fixed number of rounds of message exchange ($\lfloor(n-1)/3\rfloor+1$) is called the Immediate Consensus Algorithm (ICA). The Ensure Algorithm developed by Krings et al. [16] and the SMBTC (Synchronous Mortal Byzantine Tolerant Consensus) algorithm by Widder et al. [22] all belong to ECA. For SMC protocols, some researchers attempted to reduce the number of rounds of message exchange by employing the digital signature technology or improving the algorithm. For instance, Dalui et al. [8] introduced the Quick Consensus algorithm. A classification of consensus protocols is shown in Figure 1.

In the following sections, we will introduce the algorithms proposed to mainly reduce the rounds of message exchange required. These algorithms include the SMBTC algorithm [22], the Quick Consensus algorithm [8], and the Ensure Algorithm [16].

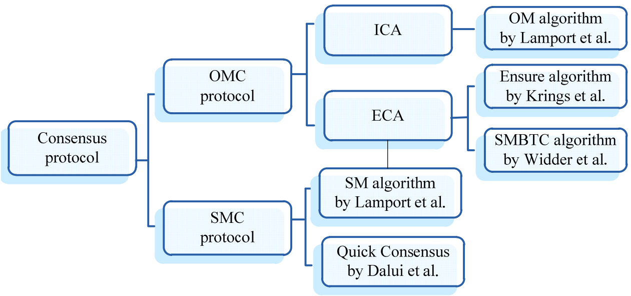

Consensus protocol
OMC protocol
SMC protocol
ICA
ECA
OM algorithm by Lamport et al.
Ensure algorithm by Krings et al.
SMBTC algorithm by Widder et al.
SM algorithm by Lamport et al.
Quick Consensus by Dalui et al.

Figure 1. A classification of consensus protocols

1.2.1. The ensure algorithm by Krings et al.

The Ensure algorithm was proposed by Krings and Feyer [16]. During execution of the Ensure algorithm, each processor p_i ($p_i \in$N, where N is the set of processors in the network, n=|N| and $i\in 1\sim n$) will check if it has collected sufficient messages after each round of message exchange. If the processor has collected sufficient messages for reaching the consensus, it will terminate the message exchange operation immediately and compute the consensus value. Through this mechanism, the Ensure algorithm can avoid unnecessary message exchange operations and reach the consensus earlier. The maximum number of faulty processors it can tolerate is $\lfloor(n-1)/3\rfloor$.

That is, the number of rounds of message exchange required by the Ensure algorithm depends on the influences, i.e. the number of actual faulty processors f, not on the total number of processors n (required rounds = $\lfloor(n-1)/3\rfloor+1$). To be succinct, the Ensure algorithm requires $min\{f+2, \lfloor(n-1)/3\rfloor+1\}$

rounds of message exchange [16]. When applied to systems with less than $\lfloor (n\text{-}1)/3 \rfloor$ faulty processors, this algorithm can effectively reduce the rounds of message exchange required. Although the Ensure algorithm can determine the number of required rounds of message exchange based on actual number of faulty processors, it still requires a large number of rounds of message exchange ($min\{f{+}2, \lfloor (n\text{-}1)/3 \rfloor{+}1\}$).

1.2.2. The SMBTC algorithm by Widder et al.

The Synchronous Mortal Byzantine Tolerant Consensus (SMBTC) algorithm was proposed by Widder, Gridling, Weiss and Blanquart [22]. During execution of the SMBTC algorithm, each processor will broadcast its initial value to all the processors and receive the initial values from other processors in the first round of message exchange. In the second round, each processor will send the initial values (a vector) obtained from other processors in the previous round to all processors in the network and receive the vector from other processors. Later, it will compare the messages sent with the messages received. If it detects any inconsistency between them, it will re-execute the algorithm. This operation continues until no inconsistency is detected. The maximum number of faulty processors this algorithm can tolerate is *n/2*.

The following is an example of "inconsistency". Suppose that five processors are given in a network, namely $p1$, $p2$, $p3$, $p4$, and $p5$. The initial values of these processors are 1, 0, 1, 1, and 0, respectively. Assume that $p4$ is a faulty processor. Thus, $p4$ may send inconsistent messages to other processors in the network. From the perspective of $p1$, this processor will send its initial value [1] to $p2$, $p3$, $p4$, and $p5$ and receive the initial values from them in the first round of message exchange. Assume that the initial values it receives from other processors are [0], [1], [1], and [0], respectively. In the second round, $p1$ will broadcast the vector consisting of initial values it has obtained in the previous round ([1][0][1][1][0]) and receive the vectors from $p2$, $p3$, $p4$, and $p5$ (as shown in Figure 2). Through comparison of the vectors, $p1$ will detect inconsistency in the 4-th column of the matrix (as shown in Figure 2) and re-execute the SMBTC algorithm until it finds no inconsistency in any column. Therefore, the SMBTC algorithm requires 2*(1 + rounds of re-executing the algorithm) rounds of message exchange, which is variable, not fixed.

However, endless re-execution of this algorithm may occur if there is always inconsistency in the matrix. To avoid this problem, Widder et al. [22] introduced the following constraint: processors with Byzantine fault will finally crash. It should be noted that this assumption does not meet the definition of the Byzantine fault. The definition of Byzantine fault is as follows: In a Byzantine fault, the behavior of a faulty component is "unpredictable" and "arbitrary". A faulty processor with Byzantine fault may lie, lose or tamper messages so it causes the most damaging type of fault and is the worst problem [1][6][23].

In this paper, we will remove the above unreasonable assumption (processors with Byzantine fault will finally crash) and revisit the consensus problem to propose a new solution.

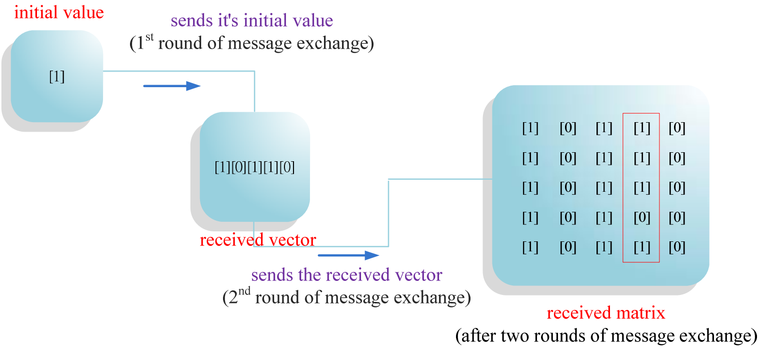


Figure 2. An example of execution of the SMBTC algorithm

1.2.3. The Quick Consensus algorithm by Dalui et al.

The Quick Consensus algorithm was proposed by Dalui et al. [8]. It is based on the SMBTC algorithm but uses digital signature to reduce the rounds of message exchange required. Hence, the Quick Consensus algorithm is also a kind of SMC protocol. In addition, this algorithm integrates the concept of grouping. It finds the optimal number of groups through mathematical analysis to reduce the rounds of message exchange required. The maximum number of faulty processors it can tolerate is n-2 and the number of rounds of message exchange is 2.

During execution of this algorithm, each processor p_i will send its initial value (v_i) after signed (denoted by $[v_i]_{p_i}$) and receive signed initial values from other processors in the first round of message exchange. In the next round, each processor p_i will sign the initial values it has received in the previous round (a vector), send them (denoted by $[v_1]_{p_1p_i}[v_2]_{p_2p_i}$, …, $[v_n]_{p_np_i}$) to other processors, and receive the vector from other processors. Through two rounds of message exchange, each processor p_i will obtain a matrix of initial values.

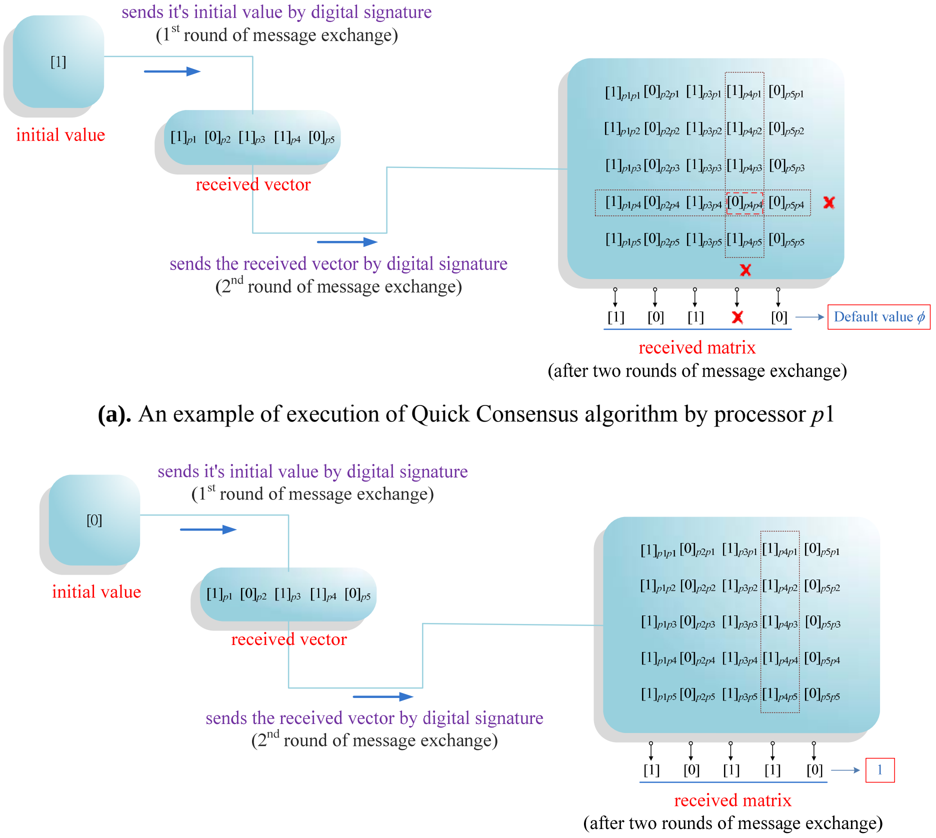
Each processor p_i will immediately check presence of any inconsistent value in the matrix and remove the values immediately. For instance, if the message from p_j is inconsistent with the messages from other processors, p_i will remove all messages related to p_j, namely $[v_1]_{p_1p_j}[v_2]_{p_2p_j}\ldots[v_n]_{p_np_j}$ and $[v_j]_{p_jp_1}[v_j]_{p_jp_2}\cdots\ldots[v_j]_{p_jp_n}$ (i.e. all values in the j-th row and the j-th column). Finally, the algorithm selects the majority value to be the consensus value.

The Quick Consensus algorithm has the "chance" to reach the consensus in merely two rounds. We quote the word "chance" because this algorithm has some drawbacks, which may make the consensus impossible to reach. Below is an example of problems which this algorithm cannot solve. A network consists of five processors $p1$, $p2$, $p3$, $p4$, and $p5$, and $p4$ is a faulty processor. We will explain the operation of the algorithm from the perspectives of $p1$ and $p2$. From the perspective of $p1$, given the initial value of 1, $p1$ will send $[1]_{p1}$ and receive the initial values from $p2$, $p3$, $p4$, and $p5$ ($[0]_{p2}$, $[1]_{p3}$, $[1]_{p4}$, and $[0]_{p5}$) in the first round. In the second round, $p1$ will broadcast the vector $[1]_{p1p1}[0]_{p2p1}[1]_{p3p1}[1]_{p4p1}[0]_{p5p1}$ and receive the vectors from $p2$, $p3$, $p4$, and $p5$ to form a vector as shown in Figure 3(a). The values in the 4-th column of the matrix are not entirely consistent, meaning that the initial value sent from $p4$ in the first round is not the same as the values from other processors. Meanwhile, the Quick Consensus algorithm will ignore all messages related to $p4$ (including values in the 4-th row and the 4-th column) and find the majority value from the remaining values. In this case, the consensus value is the default value ϕ.

From the perspective of $p2$, the message denoted by $[0]_{p4p4}$ has been signed by $p4$ two times. If $p4$ alters this value, its tampering will not be detected by other processors (for example, $p4$ sends $[0]_{p4p4}$ to $p1$ but $[1]_{p4p4}$ to $p2$). For $p2$, it has a matrix consisting of consistent values in each column (as shown in Figure 3(b)). As no message will be ignored by the algorithm, $p2$ will obtain 1 as the consensus value. In this example, the Quick Consensus algorithm is unable to make all processors reach a consensus, because non-faulty processors $p1$ and $p2$ have different consensus values.

1.3. Motivation

As afore-mentioned, the Ensure Algorithm uses oral messages and thus requires a large number of rounds of message exchange to achieve consensus. The main drawback of the SMBTC algorithm is that the number of rounds of message exchange it requires is not fixed. Besides, its assumption that all Byzantine faults will finally become crash faults is unreasonable


sends it's initial value by digital signature
(1st round of message exchange)
[1]
initial value
[1]p1 [0]p2 [1]p3 [1]p4 [0]p5
received vector
sends the received vector by digital signature
(2nd round of message exchange)
[1]p1p1 [0]p2p1 [1]p3p1 [1]p4p1 [0]p5p1
[1]p1p2 [0]p2p2 [1]p3p2 [1]p4p2 [0]p5p2
[1]p1p3 [0]p2p3 [1]p3p3 [1]p4p3 [0]p5p3
[1]p1p4 [0]p2p4 [1]p3p4 [0]p4p4 [0]p5p4 ✗
[1]p1p5 [0]p2p5 [1]p3p5 [1]p4p5 [0]p5p5
✗
[1] [0] [1] ✗ [0] → Default value φ
received matrix
(after two rounds of message exchange)

(a). An example of execution of Quick Consensus algorithm by processor $p1$

sends it's initial value by digital signature
(1st round of message exchange)
[0]
initial value
[1]p1 [0]p2 [1]p3 [1]p4 [0]p5
received vector
sends the received vector by digital signature
(2nd round of message exchange)
[1]p1p1 [0]p2p1 [1]p3p1 [1]p4p1 [0]p5p1
[1]p1p2 [0]p2p2 [1]p3p2 [1]p4p2 [0]p5p2
[1]p1p3 [0]p2p3 [1]p3p3 [1]p4p3 [0]p5p3
[1]p1p4 [0]p2p4 [1]p3p4 [1]p4p4 [0]p5p4
[1]p1p5 [0]p2p5 [1]p3p5 [1]p4p5 [0]p5p5
[1] [0] [1] [1] [0] → 1
received matrix
(after two rounds of message exchange)

(b). An example of execution of Quick Consensus algorithm by processor $p2$

Figure 3. An example of execution of Quick Consensus algorithm

and does not conform to the definition of Byzantine fault. The Quick Consensus algorithm requires only two rounds of message exchange but cannot achieve consensus in some conditions, as demonstrated in the preceding subsection. Therefore, we will revisit the consensus problem to improve the drawbacks of these algorithms. In order to reduce the complexity of message exchange, we will also employ the concept of grouping as Dalui et al. [8] did for their Quick Consensus algorithm, to find the optimal number of groups through mathematical analysis. Table 1 shows the results of previous works on the consensus problem.

Table 1. The results of previous works on the consensus problem

	Message Types		Failure Types		Comparison			
Previous Works	Oral	Signed	Dormant	Malicious	Grouping	Constraint	Rounds	Note
Lamport et al. [17]	◆			◆		$n≥3t+1$	$\lfloor(n-1)/3\rfloor+1$	
		◆		◆		$n≥t+2$	$\lfloor(n-1)/3\rfloor+1$	
Widder et al. [22]	◆		◆	◆		$n≥2t+1$	2~∞	
Dalui et al. [8]		◆	◆	◆		$n≥2t+1$	2	*
		◆	◆	◆	◆	$n≥2t+1$	4	*
Krings et al. [16]	◆		◆	◆		$n≥3t+1$	$\min\{f+2,\lfloor(n-1)/3\rfloor+1\}$	

n is the number of processors, t is the maximum number of faulty processors allowed and f is the number of actual faulty processors in the network.

*: The Quick Consensus algorithm may not reach a common consensus value.

1.4. Roadmap

This paper consists of six sections, and the remainder is organized as follows. Section 2 introduces the concept and approach. Section 3 presents the proposed protocols. Section 4 gives an example of executing the proposed protocols. Section 5 provides an analysis of the optimal number of groups. Finally, the conclusion is presented in Section 6.

2. The concept and approach

In this section, we will introduce the characteristics of the digital signature technology and the behavior of faulty processors and then explain the system model.

2.1. A brief introduction of digital signature

The increasing application of the digital signature technology to document processing and e-commerce in recent years can be attributed to five characteristics of this technology, including (1) Confidentiality: protecting confidential data from being stolen, illegally acquired or leaked; (2) Authentication: validating data sender; (3) Integrity: ensuring that data will not be tampered, resent or lost; (4) Non-repudiation: ensuring that a sender/receiver cannot deny that it has sent/received a message earlier; (5) Access control: avoiding unauthorized data access. The common encryption techniques used for signing are RSA [18] and ElGamal [10]. RSA requires that all the same messages must correspond to a specific signature. This signing method is called fixed signature. In contrast, ElGamal may encrypt a single plaintext into many possible ciphertexts. This signing technique is called probabilistic signature. The digital signature operation involves two main processes, including signing process and verification process.

Digital signature can effectively protect data from being tampered and facilitate detection of data tampering. We will integrate the digital signature technology into the proposed consensus protocol to reduce the rounds of message exchange. Thus, the proposed consensus protocol can also be classified as a SMC protocol.

2.2. The behavior of faulty processors

The behavior of a faulty processor can be classified by damaging level into two categories: dormant fault and malicious fault [4]. Dormant faults include crashes and omission. A crash fault occurs when a processor stops executing prematurely [15]. An omission fault occurs when a processor fails to send or receive a message on time or at all [20]. The malicious fault is the most damaging failure type because the behavior of a malicious processor is unpredictable and arbitrary [1][6][23]. Malicious processors may work with other faulty processors to disrupt normal operation of the system. So, this type of fault is viewed as a fault with intelligence.

2.3. System model

The main features of completely asynchronous distributed systems are: (1) no assumption on communication delays and relative speed of processors; and (2) no access to real-time clocks [11]. According to Fischer et al. [13], it is impossible to solve consensus deterministically in a completely asynchronous system. As a consequently, many researchers have proposed various ways to circumvent this limitation, including employing a partially synchronous assumption [9], using randomization protocol [14], failure detectors [2][14] or stubborn channels with a finite average response time [11]. For the above reasons, the system model of this study is built with considerations of an asynchronous network with the partially synchronous assumption.

To ensure proper operation of the system, we must set a reasonable limit on the number of faulty processors in the network. The number of faulty processors that can be tolerated in the network depends on use of digital signature. Tampering of unsigned messages is hard to detect. Thus, networks not using the digital signature technology certainly tolerate a much smaller amount of faulty processors than those using the technology. The network environment considered in this paper consists of n processors, and each of which has its initial value and always signs messages before delivering them. By using signed messages, all the processors can detect tampering of messages during message exchange. Through the time-out mechanism or encryption techniques, they can also detect loss of data packets and resend lost data immediately. Because of the characteristics of digital signature, the maximum number of faulty processors that can be tolerated in the network can be increased to n-2 [8][17]. More specifically, the number of processors in the network (n) must be greater or equal to the number of malicious processors t_m plus the number of dormant processors t_d and 2. ($n \geq t+2$ where $t = t_m+t_d$), The assumptions we have made for the asynchronous network are as follows:

- The underlying network is asynchronous with a partially synchronous assumption.
- N is the set of processors in the network, where n=|N|.
- Each processor has its own initial value and can be identified uniquely in the network.
- The failure types of the fallible processors are malicious fault and dormant fault.
- The maximum number of faulty processors allowed is n-2 ($n \geq t+2$, where $t=t_m+t_d$, t_m is the number of malicious processors and t_d is the number of dormant processors).
- A processor does not know the fault status of other processors.
- All messages are signed; processors cannot falsify a message signed by other processors.

3. The proposed protocol GCP$_{sm}$

In this section, we will introduce the proposed protocol, Grouping Consensus Protocol with signed message (GCP$_{sm}$). GCP$_{sm}$ is used to solve the consensus problem with malicious and dormant processors in the asynchronous network. The basic operation of GCP$_{sm}$ is as follows: Each group will first decide its "local" consensus value through message exchange and then elect the chief of its group. Later, all the chiefs will exchange their local consensus values with each other to obtain the "global" consensus value. After obtaining the global consensus value, they will broadcast this value among the members of their group to make all non-faulty processors reach the consensus. The proposed protocol involves three tasks, including Group_Cons Task, Gmsg_Collect Task, and Re_Elect Task. The Group_Cons Task is mainly about computation of the global consensus value. The Gmsg_Collect Task is performed to collect the messages from other groups. The Re_Elect Task is performed to re-elect the chief of the group. Upon launch, GCP$_{sm}$ will perform these three tasks concurrently. In order to effectively reduce the rounds of message exchange, we refer to the concept of grouping to group processors before executing GCP$_{sm}$. Details on how to determine the optimal number of groups will be presented in Section 5.

Because GCP$_{sm}$ derives the "global" consensus value from "local" consensus values exchanged among groups, we must obtain the "local" consensus value of each group first. We develop the SignConsensus function to meet this need. A detailed introduction of the SignConsensus function is provided in the following sections.

3.1. The SignConsensus function

The SignConsensus function consists of two tasks, including Task Cons_Compute and Task Msg_Collect. These two tasks will be run concurrently when the SignConsensus function is called. Task Msg_Collect is dealing with receiving messages from other processors in the group, and Task Cons_Compute is responsible for computing the "local" consensus value. The notations of the SignConsensus function are shown as follows:

- p_i-*Vector*(p_j) denotes the initial value from processor p_j to processor p_i.
- p_i-*Vector* denotes the received vector of processor p_i, where $p_i \in G_c$, G_c={p_{c1}, p $_{c2}$,…, p_{cn}} and p_i-*Vector*=[p_i-*Vector*(p_{c1}) …, p_i-*Vector*(p_{cn})].
- p_i-*Matrix* denotes the received matrix of processor p_i, where p_i-*Matrix* = $\begin{bmatrix} p_{c1} - Vector \\ p_{c2} - Vector \\ \cdots \\ p_{cn} - Vector \end{bmatrix}$.
- p_i-*Matrix*($p_{j,}$ •) denotes the p_j row of p_i-*Matrix*, where p_i-*Matrix*($p_{j,}$ •)=(p_i-*Matrix*($p_{j,}$ p_{c1}), p_i-*Matrix*($p_{j,}$ p_{c2}), …, p_i-*Matrix*($p_{j,}$ p_{cn}))= p_j-*Vector*.
- p_i-*Matrix*(•, p_j) denotes the p_j column of p_i-*Matrix*, where p_i-*Matrix*(•, p_j)=(p_i-*Matrix*($p_{c1,}$ p_j), p_i-*Matrix*($p_{c2,}$ p_j), …, p_i-*Matrix*($p_{cn,}$ p_j)).
- p_i-*IntrSet* denotes the set of intruders detected by processor p_i.

- p_i *_Cons* denotes the local consensus value of processor p_i.

The SignConsensus function can be presented with the following primitives and the formal description of SignConsensus function is shown in Figure 4.

- activate task(T1, T2): start the tasks T1 and T2 concurrently.
- *sign_send*(〈*msg*〉, *rcvr*): send a message 〈*msg*〉 using the digital signature technology to receiver *rcvr*.
    - *sign_send*(〈Val, p_j, v_j〉, G_c): send a Val message with the value v_j proposed by processor p_j to processors in group G_c.
    - *sign_send*(〈Vector, p_j, p_j-*Vector*〉, G_c): send a Vector message with the vector p_j-*Vector* proposed by processor p_j to processors in group G_c.
    - *sign_send*(〈Intr, p_i, p_i-*IntrSet*, *evidences*〉, G_c): send a Intr message with the set p_i-*IntrSet* and *evidences* proposed by processor p_i to processors in group G_c.
- *mark_self*(p_i-*Matrix*): mark each diagonal entities in p_i-*Matrix* to avoid the influence from self, if $j = k$ then p_i-*Matrix*(p_j, p_k)=$\bot_*$.
- *chk_dif*(p_i-*Matrix*(•, p_j)): ignore the ⊥-value, if the values in p_j column are not the same, then return "*true*"; else return "*false*".
- *decision*(p_i-*Matrix*): compute the majority value according to p_i-*Matrix*.

```
Function: SignConsensus($v_i$, $G_c$)   //for each processor $p_i$, where $p_i \in G_c$ and $v_i$ is the initial value of $p_i$

Initialization:
  activate task(Cons_Compute, Msg_Collect);
Task Cons_Compute:
/* Phase 1: Message Exchange */
//first round
  sign_send(〈Val, $p_i$, $v_i$〉, $G_c$);
  wait until (time-out interval)
    for $p_j \in G_c$ do
      if receive 〈Val, $p_j$, $v_j$〉 from $p_j$ then
        $p_i$-Vector($p_j$) = $v_j$;
      else
        $p_i$-Vector($p_j$) = $\bot_{\#}$;
    end
//second round
  sign_send(〈Vector, $p_i$, $p_i$-Vector〉, $G_c$);
  wait until (time-out interval)
    for $p_j \in G_c$ do
      if receive 〈Vector, $p_j$, $p_j$-Vector〉 then
        $p_i$-Matrix($p_j$, •) = $p_j$-Vector;
      else
        $p_i$-Matrix($p_j$, •) = ($\bot_{\#\#}$,…,$\bot_{\#\#}$);
    end
/* Phase 2: Consensus */
//mark the messages from self
  $p_i$-Matrix$_{dec}$ = mark_self($p_i$-Matrix);
    // detect the inconsistent values
  for $p_j \in G_c$ do
    if chk_dif($p_i$-Matrix$_{dec}$(•, $p_j$))=true then
      $p_i$-IntrSet = $p_i$-IntrSet ∪{$p_j$};
      $p_i$-Matrix$_{dec}$ (•, $p_j$) = ($\bot_*$,…,$\bot_*$ );
  end
  sign_send(〈Intr, $p_i$, $p_i$-IntrSet, evidences〉, $G_c$);
  wait until (time-out interval)
    for $p_j \in G_c$ do
      if receive 〈Intr, $p_j$, $p_j$-IntrSet, evidences〉 then
        $p_i$-IntrSet = $p_i$-IntrSet ∪$p_j$-IntrSet;
//compute the Consensus value
  $p_i$ _Cons = decision($p_i$-Matrix$_{dec}$);
  return ($p_i$ _Cons, $p_i$-Matrix, $p_i$-IntrSet);

Task Msg_Collect:
  when 〈Val, $p_j$, $v_j$〉 is received do
    Val_rec = Val_rec ∪{〈Val, $p_j$, $v_j$〉};
  when 〈Vector, $p_j$, $p_j$-Vector〉 is received do
    Vector_rec =Vector_rec ∪{〈Vector, $p_j$, $p_j$-Vector〉};
  when 〈Intr, $p_j$, $p_j$-IntrSet, evidences〉 is received do
    IntrSet_rec = IntrSet_rec ∪ 〈Intr, $p_j$, $p_j$-IntrSet,
                              evidences〉;
```

Figure 4. SignConsensus Function

3.1.1. The Task Cons_Compute of SignConsensus function

Task Cons_Compute has two phases, namely *The Message Exchange Phase* and *The Consensus Phase*. In the beginning, each processor in the group has an initial value. In *The Message Exchange Phase*, they will perform two rounds of message exchange and collect messages from other processors in the group through Task Msg_Collect. In the *Consensus Phase,* the local consensus value of the group will be computed. The operational process of Task Msg_Collect will be elaborated in Section 3.1.2. Below are explanations of the two phases of Task Cons_Compute.

3.1.1.1. Phase1: *The Message Exchange Phase* of Task Cons_Compute

In the first round of *The Message Exchange Phase*, each processor p_i ($p_i \in$N) will first sign its initial value and send it to other processors p_j ($j \neq i$) in the group. Meanwhile, other processors p_j will also send their signed initial values to p_i. Processor p_i will place the signed values from other processors p_j in the p_i-*Vector*(p_j). If any processor (denoted by p_k) fails to send its initial value to p_i or has a dormant fault, we use $\perp_{\#}$ to represent the missing value from p_k. In p_i-*Vector*(p_k), $\perp_{\#}$ denotes no value from p_k or p_k has a dormant fault. After the first round of message exchange, each processor p_i has a vector of values p_i-*Vector*=[p_i-*Vector*(p_{c1}) …, p_i-*Vector*(p_{cn})] (lines 2-9 in SignConsensus function).

In the second round, each processor p_i will sign the values (a vector) obtained in the previous round and send them to other processors p_j ($j \neq i$) in the group. Concurrently, other processors will also send the message (a signed vector) they have obtained to p_i. After receiving the vectors from other processors, p_i will store all the vectors, including the vector it has sent to others, in p_i-*Matrix*, which is in a matrix data structure. We use p_i-*Matrix*(p_j, •) to denote the j-th row of p_i-*Matrix*. The values in the j-th row come from the vector that p_i has received from p_j, In this round, if any processor (denoted by p_k) fails to send its vector to p_i or has a dormant fault, we use $\perp_{\#\#}$,…,$\perp_{\#\#}$ to represent the missing vector from p_k in p_i-Matrix(p_k, •). After this round of message exchange, each processor in each group will have a matrix of values (lines 10-17 in SignConsensus function). Figure 5 shows the flow chart of The Message Exchange Phase of Task Cons_Compute in the SignConsensus Function.

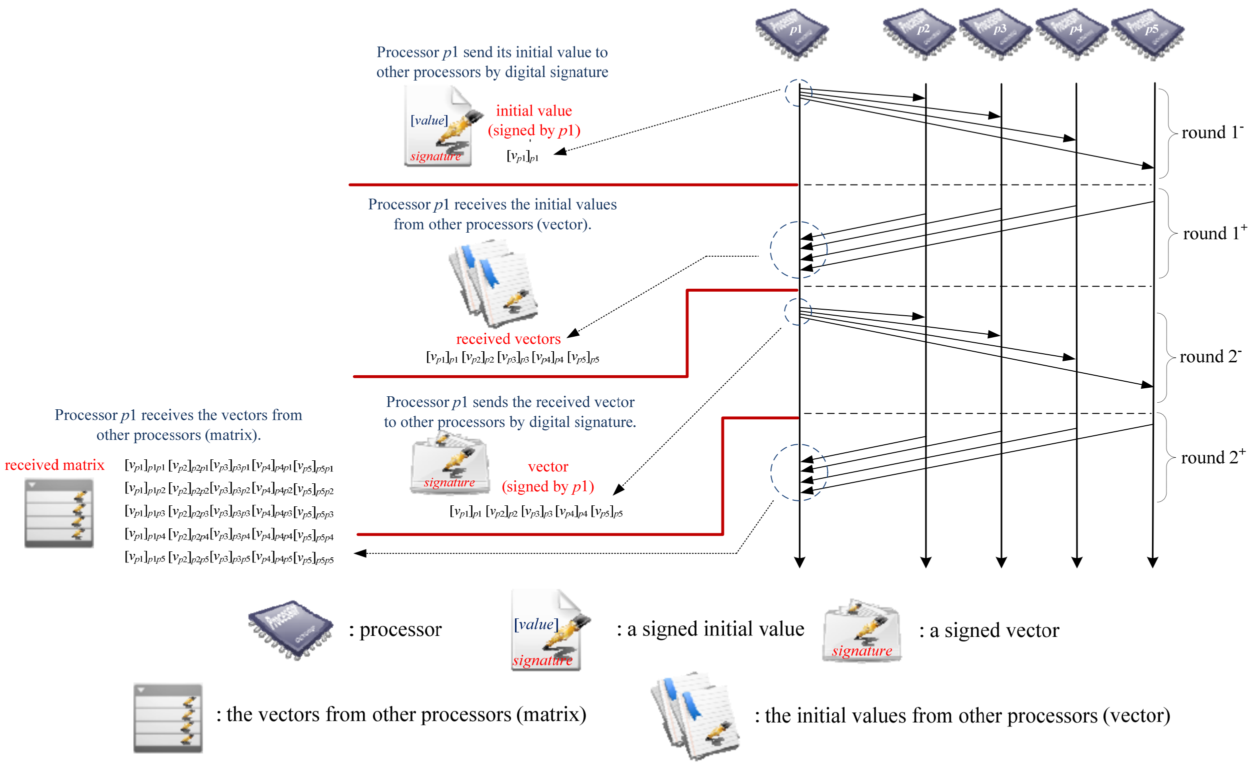

p1
p2
p3
p4
p5
Processor p1 send its initial value to other processors by digital signature
[value]
signature
initial value (signed by p1)
$[v_{p1}]_{p1}$
round 1⁻
Processor p1 receives the initial values from other processors (vector).
round 1⁺
received vectors
$[v_{p1}]_{p1}$ $[v_{p2}]_{p2}$ $[v_{p3}]_{p3}$ $[v_{p4}]_{p4}$ $[v_{p5}]_{p5}$
round 2⁻
Processor p1 sends the received vector to other processors by digital signature.
signature
vector (signed by p1)
$[v_{p1}]_{p1}$ $[v_{p2}]_{p2}$ $[v_{p3}]_{p3}$ $[v_{p4}]_{p4}$ $[v_{p5}]_{p5}$
round 2⁺
Processor p1 receives the vectors from other processors (matrix).
received matrix
$[v_{p1}]_{p1p1}$ $[v_{p2}]_{p2p1}$ $[v_{p3}]_{p3p1}$ $[v_{p4}]_{p4p1}$ $[v_{p5}]_{p5p1}$
$[v_{p1}]_{p1p2}$ $[v_{p2}]_{p2p2}$ $[v_{p3}]_{p3p2}$ $[v_{p4}]_{p4p2}$ $[v_{p5}]_{p5p2}$
$[v_{p1}]_{p1p3}$ $[v_{p2}]_{p2p3}$ $[v_{p3}]_{p3p3}$ $[v_{p4}]_{p4p3}$ $[v_{p5}]_{p5p3}$
$[v_{p1}]_{p1p4}$ $[v_{p2}]_{p2p4}$ $[v_{p3}]_{p3p4}$ $[v_{p4}]_{p4p4}$ $[v_{p5}]_{p5p4}$
$[v_{p1}]_{p1p5}$ $[v_{p2}]_{p2p5}$ $[v_{p3}]_{p3p5}$ $[v_{p4}]_{p4p5}$ $[v_{p5}]_{p5p5}$
: processor
[value]
signature
: a signed initial value
signature
: a signed vector
: the vectors from other processors (matrix)
: the initial values from other processors (vector)

Figure 5. The flow chart of *The Message Exchange Phase* of Task Cons_Compute in SignConsensus function

3.1.1.2. Phase2: *The Consensus Phase* of Task Cons_Compute

In each matrix derived through two rounds of message exchange, some messages are signed by a single processor two times (i.e. not signed by two different processors). If the processor that signs a message twice is a malicious processor, the message may have been altered, and other processors have no way to detect it (because the malicious processor owns the encryption key). To avoid this circumstance, the proposed protocol will mark diagonal entities in the matrix (i.e. replace all diagonal entities with $\perp_{*}$) before computing the consensus value (line 18 in SignConsensus function).

In the next step, each processor p_i will check consistency of values in each column. Assume that there is an inconsistent value in the j-th column. Because this value is sent by processor p_j, p_i will view p_j as a malicious processor and replace all the values in this column with $\perp_{*}$,…,$\perp_{*}$. Further, it will put p_j into p_i-*IntrSet*, which is a logbook of faulty processors it has detected (lines 19-23 in SignConsensus function), and deliver the detection result (including evidence) to other processors in the group (lines 24-28 in SignConsensus function). Finally, we will compute the local consensus value of the group using the *decision function* (lines 29-30 in SignConsensus function). The *decision function* is explained as follows:

✧ The process of the *decision function*

The decision function is intended to compute the majority value in each matrix of values. The formal description of *decision function* is shown in Figure 6. It excludes the influences from malicious processors and dormant processors before computing the majority value. First of all, it will remove ⊥-value in each column and find the index value of the column. If this value is 0, the weight (count) of 0 increases by 1 (lines 5-6 in *decision function*); if this value is 1, the weight (count) of 1 increases by 1 (lines 7-8 in *decision function*). Through this process, it can determine the majority value based on the weight (counts) of 0s and 1s. In the case that 0 and 1 have an equivalent weight (count), it uses default value ϕ as the consensus value. Here, we assume default value ϕ = 0 (lines 12-15 in *decision function*).

Function: *decision(Matrix)*

```
1. for i = 1 to cn do  //cn is the total number of rows in Matrix
2.   for j = 1 to j = cn do  //find out the non-⊥ value
3.     if Matrix(p_j, p_i) ∉ {⊥-value} do
4.       index-val = Matrix(p_j, p_i);
5.       if index-val = 0 then
6.         value0.count = value0.count + 1;
7.       elseif index-val = 1 then
8.         value1.count = value1.count + 1;
9.     break
10.    end
11.  end
12. if value0.count ≥ value1.count then
13.   return "0";
14. else
15.   return "1";
```

Figure 6. Decision function

3.1.2. The Task Msg_collect of SignConsensus Function

The Task Msg_collect is performed to collect messages from other processors in the group. Three types of messages will be collected: (1) the initial value that each processor sends in the first round (lines 31-32 in SignConsensus); (2) the vector that each processor sends in the second round (lines 33-34 in SignConsensus); and (3) the list and evidence of faulty processors that each processor has detected (lines 35-36 in SignConsensus). The pseudo code of Task Msg_collected is shown in lines 31-36 of Figure 4.

3.2. The three tasks of GCP$_{sm}$

After the SignConsensus Function, we will introduce the proposed GCP$_{sm}$ protocol. GCP$_{sm}$ performs three tasks concurrently (namely Grouping_Consensus Task, Gmsg_Collect Task, and Re_Elect Task) to compute the consensus value in the network (i.e. global consensus value). The Group_Consensus Task deals with most of the computation work, the Gmsg_Collect Task is intended to collect matrices sent by group chiefs, and the Re_Elect Task is performed to re-elect a new chief. Operational details of these three tasks are provided in Section 3.2.1, 3.2.2, and 3.2.3, respectively.

The GCP$_{sm}$ protocol can be presented with the following primitives and the formal description of GCP$_{sm}$ protocol is shown in Figure 7.

- *sign_send*(〈*msg*〉 , *rcvr*): send a message 〈*msg* 〉 using the digital signature technology to receiver *rcvr*.
    - *sign_send*(〈Mat, p$_m$, p$_m$-*Matrix*〉 , C): send a Mat message with the matrix p$_m$-*Matrix* proposed by processor p_m to C, where C is the set of chief processors.
    - *sign_send*(〈Re-elect, p$_m$, p$_j$-*Matrix*〉 , G$_j$): send a Re-elect message with the evidence p$_j$-*Matrix* proposed by processor p$_m$ to G$_j$, 〈Re-elect, p$_m$, p$_j$-*Matrix*〉 is used to request the processors in G$_j$ to re-elect the new chief, where G$_j$ is the group of p$_j$.
    - *sign_send*(〈Consensus, p$_k$, $\tilde{u}$〉 , G$_l$): send the Consensus message with the value $\tilde{u}$ proposed by processor p$_k$ to processors in group G$_l$.
- *select_chief* (Candidate$_l$): elect the processor who has the minimum ID as chief in the set of Candidate$_l$.

3.2.1. The Group_Consensus Task of GCP$_{sm}$

In the beginning of the Grouping_Consensus Task, each processor p_k can use the SignConsensus function to obtain the local consensus value (p_k*_Con*), list of faulty processors in the group (p_k-*IntrSet*), and the matrix formed after two rounds of message exchange (p_k-*Matrix*) (line 2 in GCP$_{sm}$). Later, one processor must be selected from each group to be the chief of the group. The faulty processors in each group will be first excluded (processors listed in p$_k$-*IntrSet*). The remaining processors then form a set of candidates (line 3 in GCP$_{sm}$). By using the SignConsensus function, all non-faulty processors in each group can get the same matrix, local consensus value, and the list of faulty processors in the group. Hence, any processor in the candidate set can be the chief of the group. The proposed protocol selects the processor with the minimum ID to be the chief (line 4 in GCP$_{sm}$). The selected chiefs will exchange messages with each other, and the matrix of their messages can be obtained using the SignConsensus function. In the message exchange process, each chief collects messages from other chiefs through the Gmsg_Collect Task. Details on the Gmsg_Collect Task are provided in Section 3.2.2.

In this paragraph, we will explain how GCP$_{sm}$ computes the global consensus value based on messages exchanged between chiefs. Assume that processors in group G$_l$ select p_k to be the chief (line 5 in GCP$_{sm}$). Processor p_k will send the matrix derived using the SignConsensus function to other chiefs p_j, where p$_j$ ∊ C and C is the set of chief processors. Likewise, the chiefs of other groups will also send their matrices to p_k at the same time (line 6 in GCP$_{sm}$). If the chief of group G$_j$ (denoted by p_j) does not send its matrix (Mat, p$_j$, p$_j$-*Matrix*) to p_k, Processor p_k will put p_j on its list of faulty processors and send a re-elect message to all the other processors in group G$_j$, requesting them to elect a new chief (lines 7-12 in GCP$_{sm}$). After receiving the Re-elect message, processors in group G$_j$ (excluding p_j) will run the Re_Elect Task to select a new chief among themselves. The detail description of Re_Elect Task is shown in Section 3.2.3. If all the processors in this group are faulty processors, no processor can be the chief candidate (i.e. Candidate= *null*). In this case, p_k will use the *mark_matrix* function to mark the matrix delivered by this group so that all the messages from this group will be ignored (line 14 in GCP$_{sm}$). After the message exchange operation, each chief has its own matrix and matrices from other group chiefs. The

Protocol: Grouping Consensus Protocol with signed message (GCP$_{sm}$) //for each processor p_k, where v_k is the initial value of p_k

```
Initialization:
1.  activate task(Group_Cons, Gmsg_Collect, Re_Elect);
Group_Cons Task:
/* Local Processing */
2.  (v, $p_k$-Matrix, $p_k$-IntrSet) = SignConsensus($v_k$, $G_l$);
3.  $Candidate_l = G_l - p_k$-IntrSet;
4.  $p_c$ = select_chief($Candidate_l$);
/* Group Consensus */
5.  if $p_k = p_c$ do
6.      sign_send(〈Mat, $p_k$, $p_k$-Matrix〉, C);
7.      wait until (time-out interval)
8.          for $p_j \in C$ do
9.              if doesn't receive 〈Mat, $p_j$, $p_j$-Matrix〉 from $p_j$ then
10.                 $p_k$-IntrSet = $p_k$-IntrSet ∪ {$p_j$};
11.                 if $Candidate_j \neq null$ then
12.                     sign_send(〈Re-elect, $p_k$, $p_j$-Matrix〉, $G_j$);
13.                 else
14.                     mark_matrix($p_i$-Matrix);
15.             for $p_j \in C$ do
16.                 $p_j$-$Matrix_{dec}$ = mark_self($p_j$-Matrix);
17.                 for $p_h \in G_j$ do
18.                     if chk_dif($p_j$-$Matrix_{dec}$(•, $p_h$))=true then
19.                         $p_k$-IntrSet = $p_k$-IntrSet ∪ {$p_h$};
20.                         $p_j$-$Matrix_{dec}$ (•, $p_h$) = ($\perp_*$,…,$\perp_*$);
21.                 end
22.                 matrix-val = decision($p_j$-$Matrix_{dec}$);
23.                 if matrix-val = 0 then
24.                     value0.count = value0.count + 1;
25.                 else
26.                     value1.count = value1.count + 1;
27.             end
28.         end
29.     if value0.count ≥ value1.count then
30.         sign_send(〈Consensus, $p_k$, 0〉, $G_l - p_k$-IntrSet);
31.     else
32.         sign_send(〈Consensus, $p_k$, 1〉, $G_l - p_k$-IntrSet);
33. end

Gmsg_Collect Task:
34. when 〈Mat, $p_m$, $p_m$-Matrix〉 is received do
35.     Mat_rec = Mat_rec ∪ {〈Mat, $p_m$, $p_m$-Matrix〉};
36. when 〈Re-elect, $p_m$, $p_j$-Matrix〉 is received do
37.     Re_rec = Re_rec ∪ {〈Re-elect, $p_m$, $p_j$-Matrix〉};

Re_Elect Task:
38. when 〈Re-elect, $p_e$, $p_l$-Matrix〉 is received do
39.     $p_k$-IntrSet = $p_k$-IntrSet ∪ {$p_l$};
40.     $Candidate_l = G_l - p_k$-IntrSet;
41.     $p_c$ = select_chief($Candidate_l$);
42.     goto line 5

//line 2: $p_k \in G_l$
//line 8: C is the set of chief processors
//line 11: $Candidate_j$ is the set of candidate of $G_j$, $p_j \in G_j$.
//line 38: $p_e \in C$, $p_l$ and $p_k \in G_l$
```

Figure 7. Grouping Consensus Protocol with signed message (GCP$_{sm}$)

proposed algorithm will mark the diagonal entities in each matrix and check consistency of entities in each column. Inconsistent entities will be marked, and the faulty processors will be put into p_k-*IntrSet* to preclude influences from faulty processors (lines 15-21 in GCP$_{sm}$). Later, the proposed algorithm will use the *decision function* to compute the index value of each matrix (line 22 in GCP$_{sm}$) and find the majority value. The majority value is then the global consensus value. Finally, it relies on all the chiefs to broadcast this global consensus value among non-faulty processors within their groups to achieve consensus in the network (lines 23-32 in GCP$_{sm}$).

3.2.2. The Gmsg_Collect Task of GCP$_{sm}$

The Gmsg_Collect Task is performed to collect messages from other group chiefs. Two types of messages can be collected, including (1) Mat message from other chiefs (lines 34-35 in GCP$_{sm}$) and (2) Re-elect message that requests for re-election of a new chief (lines 36-37 in GCP$_{sm}$).

3.2.3. The Re_Elect Task of GCP$_{sm}$

The Re_Elect Task is performed to elect a new chief among non-faulty processors in the group. When any processor detects that a group chief is a faulty processor, it will send a Re-elect message to all the other processors of the group, requesting them to elect a new chief. The steps are as follows: When the Re-elect message is received, p_k will immediately put the faulty processor p_l into p_k-*IntrSet* and remove it from the chief candidate set. Later, it will select a new processor from the candidate set to be the chief and return to the Group_Cons Task (lines 38-42 in GCP$_{sm}$).

4. An example of executing GCP$_{sm}$

In this section, we use an example to explain the operation of the proposed protocol. The setting of this example is as follows: A network consists of 25 processors, and each processor has an initial value. These processors are divided into five equal-sized groups, as shown in Table 2. For instance, group A comprises processors $p1$, $p2$, $p3$, $p4$, and $p5$, and their initial values are 1, 0, 0, 1, and 0, respectively. Among these processors, $p3$ and $p4$ are malicious processors. We will explain the operation of the proposed protocol from the perspective of $p1$.

Table 2. The parameters of the scenario

	Group A	Group B	Group C	Group D	Group E
group member	$p1$, $p2$, $p3$, $p4$, $p5$	$p6$, $p7$, $p8$, $p9$, $p10$	$p11$, $p12$, $p13$, $p14$, $p15$	$p16$, $p17$, $p18$, $p19$, $p20$	$p21$, $p22$, $p23$, $p24$, $p25$
initial values	1,0,0,1,0	1,0,0,0,0	1,1,1,1,1	1,0,1,1,1	1,1,0,1,1
malicious processor	$p3$, $p4$		$p11$, $p15$	$p16$	$p21$, $p22$

4.1. Local processing

In the beginning, $p1$ uses the SignConsensus function to obtain the local consensus value of the group (group A). This operation requires two rounds of message exchange. In the first round, $p1$ will sign its initial value $[1]_{p1}$ and send it to other processors within the group. At the same time, it will also receive signed initial values from $p2$, $p3$, $p4$, and $p5$. These signed values are, respectively, denoted by $[0]_{p2}$, $[0]_{p3}$, $[1]_{p4}$, and $[0]_{p5}$. Hence, $p1$ can obtain a vector consisting of $[1]_{p1}[0]_{p2}[0]_{p3}[1]_{p4}[0]_{p5}$. It will store these values into p_1-*Vector*=[p_1-*Vector*(p_1) …, p_1-*Vector*(p_5)], as shown in Figure 8(a). In the second round, $p1$ will sign the vector it has obtained in the previous round and send it to other processors. At the same time, it will also receive signed vectors from other processors. After two rounds of message exchange, each processor can obtain a matrix of messages. As shown in Figure 8(b), $p1$ can obtain a matrix consisting of the vector $[1_{p1p2}\ 0_{p2p2}\ 1_{p3p2}\ 1_{p4p2}\ 0_{p5p2}]$ from $p2$, the vector $[1_{p1p3}\ 0_{p2p3}\ 1_{p3p3}\ 0_{p4p3}\ 0_{p5p3}]$ from $p3$, the vector $[1_{p1p4}\ 0_{p2p4}\ 0_{p3p4}\ 0_{p4p4}\ 0_{p5p4}]$ from $p4$, and the vector $[1_{p1p5}\ 0_{p2p5}\ \perp_{\#p5}\ \perp_{\#p5}\ 0_{p5p5}]$ from $p5$.

In the next step, $p1$ will mark all the diagonal entities in the matrix with $\perp_*$ and check consistency of entities in each column. In this example, it will detect the inconsistency in the 3rd and 4th columns. Take the inconsistency in the third column as an example. The entity in row 1 and column 3 is 0, the entity in row 2 and column 3 is 1, and the entity in row 4 and column 3 is 0. It can detect that $p3$ and $p4$ are faulty processors. Hence, $p1$ will put $p3$ and $p4$ in p_1-*IntrSet* and use $\perp_*,\ldots,\perp_*$ to take the place of values in the third and fourth column. Besides, it will also deliver the updated p_1-*IntrSet* to other processors in the group to tell them that $p3$ and $p4$ are faulty processors. Later, it will use the *decision function* to compute the local consensus value. The computation process is as follows: First, it finds the index value of each column. The valid index values are 0 and 1. The index value of column 1 is 1, the index value of column 2 is 0, and the index value of column 5 is 0. As shown in Figure 8(c), because 0 is the majority value among 1, 0, and 0, the local consensus value of group A is 0.

In the following step, a chief should be selected from non-faulty processors in each group. The processor with the minimum ID can be the chief of its group. Take group A as an example. Excluding faulty processors $p3$ and $p4$, $p1$ is the non-faulty processor with the minimum ID. Hence, $p1$ is selected to be the chief of group A. It will perform message exchange on behalf of group A to obtain the global consensus value. Meanwhile, all the other groups will also compute their local consensus values and select their chiefs. In this example, there is no faulty processor in group B. As $p6$ is the non-faulty processor with the minimum ID, $p6$ is selected to be the chief of group B. Subgroup C has two faulty processors, namely $p11$ and $p15$, so $p12$ is selected to be the chief. Subgroup D has one faulty processor, $p16$, so $p17$ is selected. Subgroup E has two faulty processors, $p21$ and $p22$. $p23$ will be selected be the chief of this group.

4.2. Group Consensus

In the Group Consensus phase, all the group chiefs will exchange messages to derive the global consensus value. Chiefs $p1$, $p6$, $p12$, $p17$, and $p23$ will exchange their Mat messages. Figure 9(a) shows the exchange of Mat messages between $p1$ and other chiefs. After exchanging Mat messages, $p1$ will have five matrices, including $p6$, $p12$, $p17$, $p23$, and itself, as shown on the top of Figure 9(b). It will mark the diagonal entities in all these matrices and check consistency of values in each column. As shown in the center of Figure 9(b), inconsistent values will be marked. Later, $p1$ will use the *decision function* to compute the index value of each matrix.

In this example, the index values of the five matrices are 0, 0, 1, 1, and 1, respectively. Based on the majority rule, $p1$ will find 1 as the global consensus value, as shown in the bottom of Figure 9(b). Finally, $p1$ will send this global consensus value to other non-faulty processors in the group. While $p1$ is broadcasting the global consensus value, other chiefs are also broadcasting the global consensus values they obtain among their peer non-faulty processors. Therefore, all non-faulty processors in the network will end up having a consensus value.

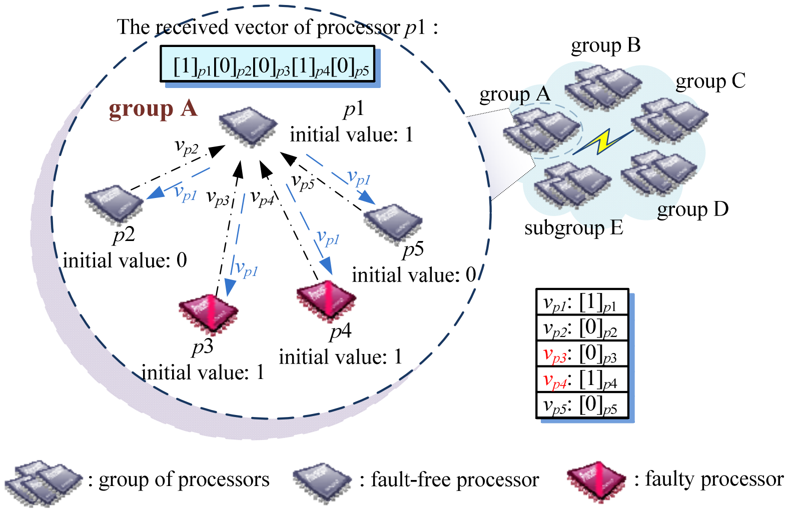
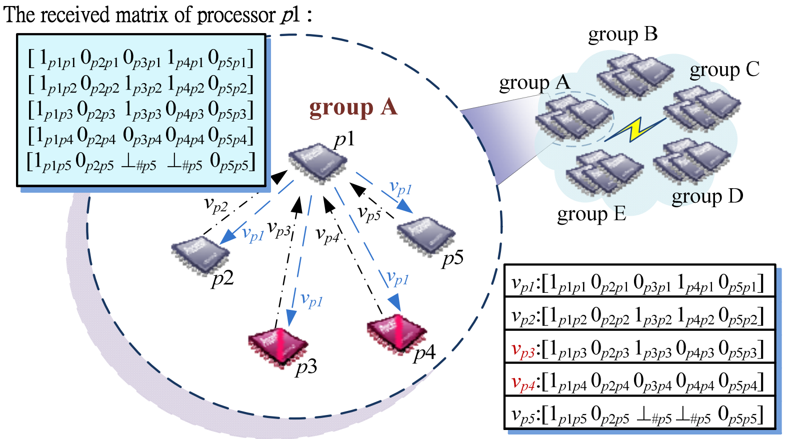
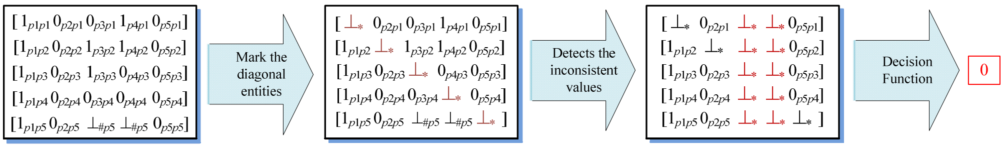


Figure 8 (a). The 1st round of *Message Exchange* phase of $p1$ in group A (SignConsensus function)



The received matrix of processor $p1$:
$[1_{p1p1}\ 0_{p2p1}\ 0_{p3p1}\ 1_{p4p1}\ 0_{p5p1}]$
$[1_{p1p2}\ 0_{p2p2}\ 1_{p3p2}\ 1_{p4p2}\ 0_{p5p2}]$
$[1_{p1p3}\ 0_{p2p3}\ 1_{p3p3}\ 0_{p4p3}\ 0_{p5p3}]$
$[1_{p1p4}\ 0_{p2p4}\ 0_{p3p4}\ 0_{p4p4}\ 0_{p5p4}]$
$[1_{p1p5}\ 0_{p2p5}\ \perp_{\#p5}\ \perp_{\#p5}\ 0_{p5p5}]$
group A
$p1$
v_{p2}
v_{p1}
v_{p3}
v_{p4}
v_{p5}
$p2$
$p3$
$p4$
$p5$
group B
group A
group C
group D
group E
v_{p1}:$[1_{p1p1}\ 0_{p2p1}\ 0_{p3p1}\ 1_{p4p1}\ 0_{p5p1}]$
v_{p2}:$[1_{p1p2}\ 0_{p2p2}\ 1_{p3p2}\ 1_{p4p2}\ 0_{p5p2}]$
v_{p3}:$[1_{p1p3}\ 0_{p2p3}\ 1_{p3p3}\ 0_{p4p3}\ 0_{p5p3}]$
v_{p4}:$[1_{p1p4}\ 0_{p2p4}\ 0_{p3p4}\ 0_{p4p4}\ 0_{p5p4}]$
v_{p5}:$[1_{p1p5}\ 0_{p2p5}\ \perp_{\#p5}\ \perp_{\#p5}\ 0_{p5p5}]$

Figure 8 (b). The 2nd round of *Message Exchange* phase of $p1$ in group A (SignConsensus function)



$[1_{p1p1}\ 0_{p2p1}\ 0_{p3p1}\ 1_{p4p1}\ 0_{p5p1}]$
$[1_{p1p2}\ 0_{p2p2}\ 1_{p3p2}\ 1_{p4p2}\ 0_{p5p2}]$
$[1_{p1p3}\ 0_{p2p3}\ 1_{p3p3}\ 0_{p4p3}\ 0_{p5p3}]$
$[1_{p1p4}\ 0_{p2p4}\ 0_{p3p4}\ 0_{p4p4}\ 0_{p5p4}]$
$[1_{p1p5}\ 0_{p2p5}\ \perp_{\#p5}\ \perp_{\#p5}\ 0_{p5p5}]$
Mark the diagonal entities
$[\perp_{*}\ 0_{p2p1}\ 0_{p3p1}\ 1_{p4p1}\ 0_{p5p1}]$
$[1_{p1p2}\ \perp_{*}\ 1_{p3p2}\ 1_{p4p2}\ 0_{p5p2}]$
$[1_{p1p3}\ 0_{p2p3}\ \perp_{*}\ 0_{p4p3}\ 0_{p5p3}]$
$[1_{p1p4}\ 0_{p2p4}\ 0_{p3p4}\ \perp_{*}\ 0_{p5p4}]$
$[1_{p1p5}\ 0_{p2p5}\ \perp_{\#p5}\ \perp_{\#p5}\ \perp_{*}]$
Detects the inconsistent values
$[\perp_{*}\ 0_{p2p1}\ \perp_{*}\ \perp_{*}\ 0_{p5p1}]$
$[1_{p1p2}\ \perp_{*}\ \perp_{*}\ \perp_{*}\ 0_{p5p2}]$
$[1_{p1p3}\ 0_{p2p3}\ \perp_{*}\ \perp_{*}\ 0_{p5p3}]$
$[1_{p1p4}\ 0_{p2p4}\ \perp_{*}\ \perp_{*}\ 0_{p5p4}]$
$[1_{p1p5}\ 0_{p2p5}\ \perp_{*}\ \perp_{*}\ \perp_{*}]$
Decision Function
0

Figure 8(c). The *Consensus* phase of $p1$ in group A (SignConsensus function)

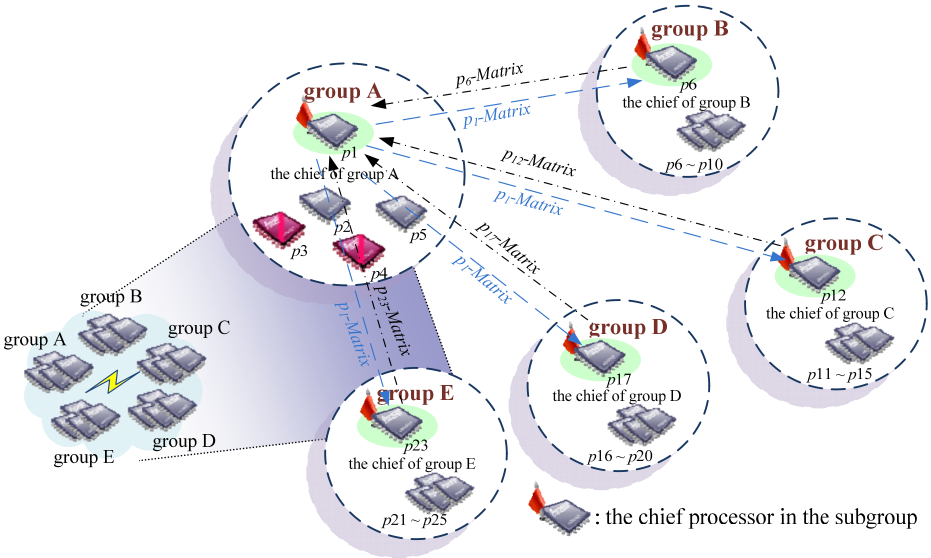


Figure 9(a). The message exchange of Mat message between the chiefs.

5. The Analysis of the Number of Groups

In this section, we will perform mathematical analysis to find the optimal number of groups for GCP$_{sm}$. GCP$_{sm}$ involves *the local processing phase* and *the global processing phase.* The number of rounds of message exchange that each phase requires is as follows:

Local processing: $(n/g)((n/g)\text{-}1)g+(n/g)((n/g)\text{-}1)g+(n/g)((n/g)\text{-}1)g$ (1)

Global processing: $g(g\text{-}1)+(g\text{-}1)(n/g)(n/g)(g)\text{-}2(n/g)(g)+(n\text{-}g\text{-}(n\text{-}2))$ (2)

Given n processors in a network, we divide these n processors into g equal-sized groups. In other words, each group has n/g processors. In the *local processing phase*, each processor needs to perform two rounds of message exchange. In the first round of message exchange, each processor (n/g in total) in each group will send their signed initial values to other processors in the group ((n/g)-1 in total) and also receive signed values from them. Therefore, a total of (n/g)((n/g)-1) times of message exchange will be performed in the first round. In the second round, each processor will send the vector it has obtained in the previous round to (n/g)-1 other processors in the group and also receive the vectors from them. A total of (n/g)((n/g)-1) times of message exchange will be needed. Later, each processor will check consistency of obtained values to find faulty processors and send the detection result to other processors in the group. A total of (n/g)((n/g)-1) times of message exchange will also be required. In sum, because each processor in each group has to perform two rounds of message exchange and send fault detection results, the entire network needs to perform (n/g)((n/g)-1)g+(n/g)((n/g)-1)g+(n/g)((n/g)-1)g times of message exchange in the *local processing phase.*

In the *global processing phase*, because each of the g group chiefs will exchange messages with g-1 chiefs, g(g-1) times of message exchange are needed. If any chief is found to be a faulty processor, a Re-elect message will be sent to all the processors in its group, requesting them to select a new chief. In other words, if any chief is a faulty processor, g-1 chiefs will send a total of n/g Re-elect messages to processors in its group. If all the n/g processors in this group become faulty while being the chief of their group, the above operation will be executed n/g times. As there are g groups and at most n-2 faulty processors in the network, GCP$_{sm}$ has to send the Re-elect message (g-1)(n/g)(n/g)(g)-2(n/g)(g) times in the worst case. After the global consensus value is obtained, all the chief processors have to broadcast this value to other non-faulty processors in their groups. In the worst case, this value has to be sent (n-g-(n-2)) times. Therefore, the entire network needs to perform g(g-1)+(g-1)(n/g)(n/g)(g)-2(n/g)(g)+(n-g-(n-2)) times of message exchange in the *global processing phase.*

The total number of rounds of message exchange that GCP$_{sm}$ requires is as shown in Equation (3) (the sum of Equation (1) and Equation (2)). We conduct the following mathematical analysis to find the best number of groups.

$$f = g^2+2n^2/g\text{-}2g+n^2\text{-}5n+2 \qquad (3)$$

$$\frac{\partial f}{\partial g} = 2g - \frac{2n^2}{g^2} - 2 \qquad (4)$$

$$\frac{\partial^2 f}{\partial g^2} = \frac{4n^2}{g^3} + 2 > 0 \qquad (5)$$

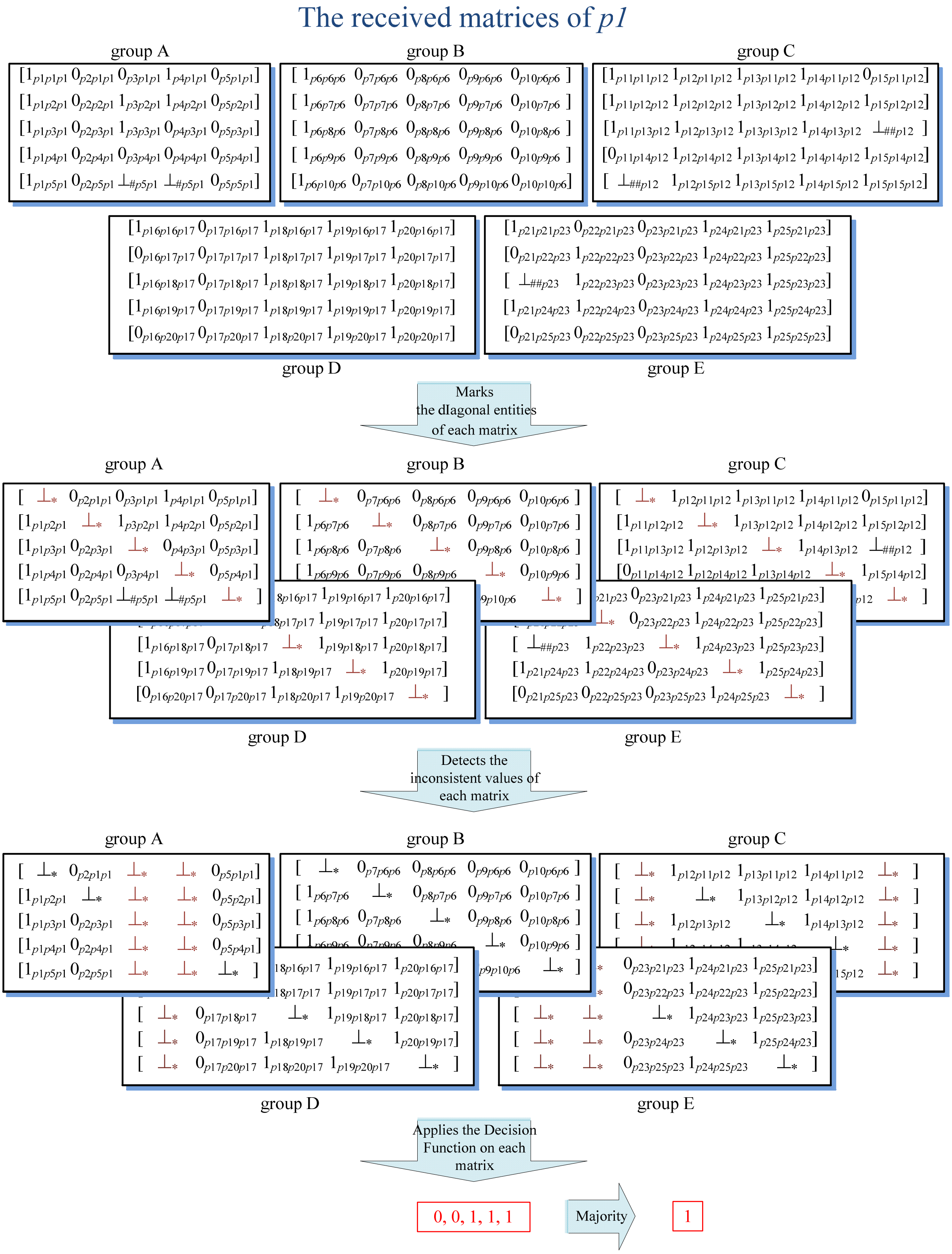

The received matrices of p1
group A
group B
group C
group D
group E
Marks the dIagonal entities of each matrix
Detects the inconsistent values of each matrix
Applies the Decision Function on each matrix
0, 0, 1, 1, 1
Majority
1

Figure 9(b). An example of computing the global consensus value (from the perspective of $p1$)

Equation (5) is the result of quadratic partial differentiation. Both n and g are greater than zero, so Equation (5) must be greater than zero. We can infer that when Equation (4) (ordinary differentiation) is zero, the obtained g can result in a minimum f. In other words, the best number of groups for a network consisting of n processors is $g \cong \frac{1}{3}\sqrt[3]{19n^2}$. This grouping method can minimize the complexity of message exchange required by the proposed protocol.

6. Conclusions

As mentioned earlier, OMC protocols exchange oral messages to achieve consensus and therefore require a large number of rounds of message exchange. The OM algorithm, the Ensure algorithm, and the SMBTC algorithm are all OMC protocols. SMC protocols exchange signed messages to achieve consensus. Due to the characteristics of digital signature, they require a significantly smaller number of rounds of message exchange. The SM algorithm and the Quick Consensus algorithm belong to SMC protocols. However, the existing signed message-based consensus algorithms still have some drawbacks which make them ineffective in some specific conditions. Therefore, we revisited the consensus problem in distributed systems in this paper. We proposed a signed message-based consensus algorithm that makes use of digital signature to effectively reduce the complexity of message exchange. In addition, we integrated the concept of grouping into the algorithm and obtained the optimal number of groups by mathematical analysis. That is, the proposed algorithm can not only facilitate achievement of consensus by removing influences from malicious and dormant processors but also reduce the complexity of message exchange.

References

[1] **A.N. Bessani, M. Correia, J. da Silva Fraga, L.C. Lung.** An Efficient Byzantine-Resilient Tuple Space. In: *IEEE Transactions on Computers*, Vol. 58, No. 8, pp. 1080-1094, 2009. http://dx.doi.org/10.1109/TC.2009.71.

[2] **T.D. Chandra, S. Toueg**. Unreliable Failure Detectors for Reliable Distributed Systems. In: *Journal of the ACM*, vol.43, no.2, pp.225-267, 1996. http://dx.doi.org/10.1145/226643.226647.

[3] **C.F. Cheng, S.C. Wang, T. Liang**. Byzantine Agreement Protocol & Fault Diagnosis Agreement for Mobile Ad-Hoc Network. In: *Fundamenta Informaticae*, vol. 89, no. 2, pp.161-187, 2008.

[4] **C.F. Cheng, S.C. Wang and T. Liang.** Investigation of Consensus Problem over Combined Wired/Wireless Network. In: *Journal of Information Science and Engineering*, vol. 25, no. 4, pp.1267-1281, 2009.

[5] **C.F. Cheng, S.C. Wang, T. Liang.** File Consistency Problem of File-Sharing in Peer-to-Peer Environment. In: *International Journal of Innovative Computing, Information and Control*, vol. 6, no. 2, pp.601-613, 2010.

[6] **M. Correia, N. F. Neves, L. C. Lung, P. Verissimo.** Worm-IT – A Wormhole-based Intrusion-tolerant Group Communication System. In: *Journal of Systems and Software*, vol. 80, no. 2, pp.178–197, 2007. http://dx.doi.org/10.1016/j.jss.2006.03.034.

[7] **M. Correia, N. F. Neves, P. Verissimo.** From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. In: *The Computer Journal*, vol.49, no.1, pp.82-96, 2006. http://dx.doi.org/10.1093/comjnl/bxh145.

[8] **M. Dalui, B. Chakraborty, B.K. Sikdar.** Quick Consensus Through Early Disposal of Faulty Processes. In: *Proceedings of the 22nd IEEE International Conference on Systems, Man and Cybernetics*. pp. 1989-1994, 2009.

[9] **C. Dwork, N. Lynch, L. Stockmeyer.** Consensus in the Presence of Partial Synchrony. In: *Journal of the ACM*, vol. 35, no. 2, pp.288-323, 1988. http://dx.doi.org/10.1145/42282.42283.

[10] **T. Elgamal.** A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp.469-472, 1985. http://dx.doi.org/10.1109/TIT.1985.1057074.

[11] **C. Fetzer, U. Schmid, M. Susskraut.** On the Possibility of Consensus in Asynchronous Systems with Finite Average Response Times. In: *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, 2005.

[12] **M.J. Fischer, N. A. Lynch.** A Lower Bound for the Time to Assure Interactive Consistency. In: *Information Processing Letters*, vol.14, no.3, pp.183-186, 1982. http://dx.doi.org/10.1016/0020-0190(82)90033-3.

[13] **M.J. Fischer, N.A. Lynch, M.S. Paterson.** Impossibility of Distributed Consensus with One Faulty Process. In: *Journal of the ACM*, vol. 32, no. 2, pp.374-382, 1985. http://dx.doi.org/10.1145/3149.214121.

[14] **R. Friedman, A. Mostefaoui, M. Raynal.** Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Byzantine Systems. In: *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp.46-56, 2005. http://dx.doi.org/10.1109/TDSC.2005.13.

[15] **S. Jafar, A. Krings, T. Gautier.** Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing. In: *IEEE Transactions on Dependable and Secure Computing*, Vol. 6, No. 1, pp. 32-44, 2009.

[16] **A. W. Krings, T. Feyer.** The Byzantine Agreement Problem: Optimal Early Stopping. In: *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, LNCS 520, Springer-Verlag, Berlin, 1999.

[17] **L. Lamport, R. Shostak, M. Pease.** The Byzantine Generals Problem. In: *ACM Transaction on Programming Language Systems*, Vol.4, No. 3, pp.382-401, 1982. http://dx.doi.org/10.1145/357172.357176.

[18] **R.L. Rivest, A. Shamir, L. Adleman.** A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In: *Communications of the ACM*, vol. 21, no. 2, pp.120–126, 1978. http://dx.doi.org/10.1145/359340.359342.

[19] **A. Silberschatz, P. B. Galvin, G. Gagne.** In: *Operating System Concepts*, 8th Edition, John Wiley & Sons Inc, 2008.

[20] **H.Y. Tzeng, K.Y. Siu.** On the Message and Time Complexity of Protocols for Reliable Broadcasts/Multicasts in Networks with Omission Failures. In: *IEEE Journal on Selected Areas in Communications*, Vol. 13, No. 7, pp. 1296-1308, 1995. http://dx.doi.org/10.1109/49.414647.

[21] **S.C. Wang, K.Q. Yan, C.F. Cheng.** Asynchronous Consensus Protocol for the Unreliable Un-fully Connected Network. In: *ACM Operating Systems*

Review, vol. 37, no. 3, pp. 43-54, July 2003, http://dx.doi.org/10.1145/881783.881789.

[22] J. Widder, G. Gridling, B. Weiss, J.P. Blanquart. Synchronous Consensus with Mortal Byzantines. In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable systems and Networks*, pp.102-112, 2007.

[23] W. Zhao. Design and Implementation of a Byzantine Fault Tolerance Framework for Web Services. In: *Journal of Systems and Software*, vol. 82, no. 6, pp.1004–1015, 2009. http://dx.doi.org/10.1016/j.jss.2008.12.037.

Received August 2011.