Global data computation in chordal rings

Xianbing Wang\textsuperscript{a}, Yong Meng Teo\textsuperscript{b,c,}\textsuperscript{*}

\textsuperscript{a} Computer Center, Wuhan University, Wuhan 430072, China
\textsuperscript{b} Singapore-Massachusetts Institute of Technology Alliance, Singapore 117576, Singapore
\textsuperscript{c} Department of Computer Science, National University of Singapore, Singapore 117417, Singapore

A R T I C L E   I N F O

Article history:
Received 23 August 2006
Received in revised form 17 March 2009
Accepted 4 April 2009
Available online 21 April 2009

Keywords:
Global data computation
Chordal rings
Perfect failure detector
Non-round-based protocol

A B S T R A C T

Existing Global Data Computation (GDC) protocols for asynchronous systems are round-based algorithms designed for fully connected networks. In this paper, we discuss GDC in asynchronous chordal rings, a non-fully connected network. The virtual links approach to solve the consensus problem may be applied to GDC for non-fully connected networks, but it incurs high message overhead. To reduce the overhead, we propose a new non-round-based GDC protocol for asynchronous chordal rings with perfect failure detectors. The main advantage of the protocol is that there is no notion of rounds. Every process creates two messages initially, with one message traversing in a clockwise direction and visiting each and every process in the chordal ring. The second message traverses in a counterclockwise direction. When there is direct connection between two processes, a message is sent directly. Otherwise, the message is sent via virtual links. When the two messages return, the process decides according to the information maintained by the two messages. The perfect failure detector of a process need only detect the crash of neighboring processes, and the crash information is disseminated to all other processes. Analysis and comparison with two virtual links approaches show that our protocol reduces message complexity significantly.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

1.1. GDC & related work

In distributed computation, a Global Data (GD) item is a vector with one entry that is filled with an appropriate value proposed by a corresponding process. The problem of computing global data and providing each process with a copy defines the Global Data Computing (GDC) problem [8,12]. This approach can be used in Global Function Computation [11,13], which is a key component for solving many distributed computing problems. Global Function Computation consists of: (1) requiring processes to define a global data item; (2) computing a deterministic function of this data item; and (3) providing each process with the corresponding result. The Atomic Commitment problem [3] is an example of Global Function Computation.

The GDC problem may be more precisely defined as follows: Assume a distributed system with $n$ processes, where $GD[0..n-1]$ is a vector global data item with one entry per process, and $v_i$ denotes the value provided by $p_i$ to fill the global data entry. The GDC problem consists of building $GD$ and providing a copy to each process. Let $GD_i$ denote the local variable of $p_i$ that contains the local copy of $GD$, and $\bot$ be a default value that is used instead of the value $v_i$ when the corresponding process $p_i$ fails prematurely. The problem is formally specified by a set of four properties as follows [12]:

\begin{itemize}
  \item Termination: Eventually, every correct process $p_i$ decides on a local vector $GD_i$.
  \item Validity: No spurious initial value exists. $\forall i$: if $p_i$ decides on $GD_i$, then $\forall j \in \{v_i, \bot\}$.
  \item Agreement: No two processes decide on different Global Data. $\forall i, j$: if $p_i$ decides on $GD_i$ and $p_j$ decides on $GD_j$, then $\forall k \in \{GD_i[k], GD_j[k]\}$.
  \item Obligation: If a process decides, its initial value belongs to the Global Data. $\forall i$: if $p_i$ decides on $GD_i$, then $GD_i[i] = v_i$.
\end{itemize}

According to the well-known FLP consensus\textsuperscript{1} impossibility result [10], the GDC problem has no deterministic solution in an asynchronous distributed system with crash failures. To circumvent the impossibility problem, perfect failure detectors, as defined by Chandra and Toueg [5], may be used in a GDC protocol.

---

\textsuperscript{1} A consensus problem is for a group of processes to achieve agreement on a proposed value in distributed systems with failures, and a GDC problem can be regarded as a special consensus problem where processes are required to agree on the global data.

---

0743-7315/$-$ see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.04.007
for crash failures to be tolerated in asynchronous distributed systems [12]. Here, when a process crashes, it stops prematurely and remains crashed forever. The GDC protocol proposed in [12] terminates in \(\min(2f + 2, t + 1)\) asynchronous computation rounds, where at most \(t (t < n)\) processes can crash and \(f\) is the number of processes that actually crash in an execution. The GDC protocol is extended to improve the lower bound to \(f + 2, t + 1, n\) in [8]. However, both GDC protocols assume that a reliable communication channel connects each pair of processes. Essentially, an assumption is made that the distributed systems are fully connected.

As in [8,12], this paper discusses the GDC problem based on perfect failure detectors. It is well known that perfect failure detectors are neither implementable in truly asynchronous distributed systems or in partially synchronous distributed systems [16]. However, perfect failure detectors can be implemented in some asynchronous distributed systems with certain restrictions [4]. Examples include timed asynchronous distributed systems [7,9] and asynchronous systems with link-related assumptions [20]. This paper discusses the GDC problem in asynchronous distributed systems where perfect failure detectors can be implemented.

1.2. Motivation

In this paper, we show the first step in solving the GDC problem in distributed systems with a non-fully connected chordal rings topology. Chordal rings have been studied for many years as the interconnection architecture for parallel and distributed systems [12,14], and current work includes [15,21,22]. An attractive property of chordal rings is that they have Hamiltonian cycles built in and they are readily visible. In other networks, researchers have to go to great length to establish Hamiltonicity. Other properties include symmetry, ease of routing, and robustness. Although the latter advantages are not unique to chordal rings, not many networks offer all these desirable properties at the same time [21].

Chordal rings introduce link redundancy to improve reliability. With alternate paths between processors, the network can tolerate several processors and links failures. A chordal ring \(G_{nc} = (V, E)\), consisting of \(n\) processors \(\{p_0, p_1, \ldots, p_{n-1}\}\) and a chord structure \((d_1, d_2, \ldots, d_k)\), in which each processor is connected to other processors at distance \(d_1, d_2, \ldots, d_k, n-d_k, \ldots, n-d_2, n-d_1\), (calculated in a clockwise direction), remains a connected graph when any \(2k + 1\) nodes are deleted [27]. If virtual links for each pair of processors in non-fully connected chordal rings can be established, the GDC problem in chordal rings can be solved using round-based GDC protocols [12,8], and the chordal ring \(G_{nc} = (d_1, d_2, \ldots, d_k)\) can tolerate up to \(2k + 1\) crashes.

A shortcoming of the virtual links approach is high message overhead cost. In each round, the number of messages increases significantly because the method relies on the use of node-disjoint paths. In this paper, we also discuss the other important issue of how to implement failure detectors. There are two main methods: using virtual links for each pair of processors to detect each other’s crash by adopting the traditional implementation scheme [5], or using gossip-style failure detection [23], where a process detecting a neighbor’s crash multicasts the crash information to all its neighbors. Clearly, the first method incurs more redundant messages.

1.3. Contribution

To reduce message overhead, we propose a non-round-based GDC protocol for asynchronous chordal rings. In this protocol, every process initially creates two messages, with one message traversing in a clockwise direction and visiting each and every process in the chordal ring, and the second message traversing in a counterclockwise direction. When there is direct connection between two processes, a message is sent directly. Otherwise, the message is sent via a virtual link. When the two messages return, the process makes a decision based on the information maintained by the two messages. A perfect failure detector module
in each process detects the crash of its neighboring processes, and the crash information is disseminated to all the other processes. We adopt gossip-style failure detection. The non-round-based GDC protocol was first introduced in [26] for fully connected asynchronous distributed systems with perfect failure detectors. In contrast, we propose in this paper a non-round-based algorithm for non-fully connected networks.

Analysis and comparison with virtual links approaches show that our protocol reduces message complexity significantly. In addition, we have investigated how chordal rings affect the message efficiency of our proposed protocol. The result shows that with our protocol, \( C_n(2, \ldots, k + 1) \) is the most message efficient chordal ring.

1.4. Roadmap

This paper consists of six sections. Section 2 introduces the system model. Section 3 describes our proposed non-round GDC protocol for asynchronous chordal rings with perfect failure detectors. In Section 4, we present the correctness proof of the proposed protocol. Section 5 analyzes message complexity and compares our approach with other virtual links approaches. The section also discusses the impact of chordal rings on message complexity. Section 6 concludes the paper.

2. System model

An asynchronous chordal ring \( C_n(d_1, d_2, \ldots, d_k) \) consisting of \( n \) processors, \( \prod \mapsto \{p_0, \ldots, p_{n-1}\} \), tolerates at most \( 2k + 1 \) failures where \( 0 \leq k < (n - 1)/2 \). When \( k \) is 0, the chordal ring is \( C_n^1 \). There is no shared memory in the system, and processes exchange messages through ring edges and channels. Channels are reliable, i.e., there are no spurious messages, no loss and no corruption, but there is no need for messages to be sent and received in FIFO order. Processes \( p_i \) and \( p_j \) are neighbors if they are connected by a ring edge or a chord. We denote the set of all neighbors of \( p_i \) as \( \prod \). In a truly asynchronous distributed system, process speed and communication delay are arbitrary. We consider, in this paper, a more restricted asynchronous system where perfect failure detectors defined by the following properties can be implemented [5]:

- Completeness: Eventually, every crashed process is suspected by every correct process.
- Accuracy: No process is suspected before it crashes.

The implementation of perfect failure detectors is discussed in [9,20,24]. For example, in timed asynchronous distributed systems [7], there exists no upper bound on the transmission delay of messages or on execution time. Also, processors can become transiently or permanently partitioned. Each processor has an unsynchronized local hardware clock to measure time intervals within a known maximum error. To implement perfect failure detectors [9], the timed asynchronous distributed system model introduces hardware watchdogs. This enables a non-responsive processor to be reset as soon as the value of the hardware clock reaches a certain value, and then ensures that a processor crashes before it is suspected. An example is Sabel and Marzullo’s proposed simulating fail-stop in asynchronous distributed systems [24]. The accuracy property of failure detectors is relaxed using the happens-before relation to simulate a perfect failure detector in a truly asynchronous distributed system. If all processors participate in the protocol proposed in [24] and there are no hidden channels, the protocol ensures that a processor crashes before it is suspected.

In this paper, we adopt the crash failure model. When a process crashes, it definitely stops its activity and does nothing else [17]. A process is correct if it does not crash till the end of execution.

In asynchronous chordal rings, each process \( p_i \) is equipped with a failure detector module. The module provides \( p_i \) with a set of variables called suspected, that contains the identities of neighbors that are assumed to have crashed. Process \( p_i \) can only read this set of variables, which is continuously updated by the module. If \( p_j \in \text{suspected}_i \), we say that “\( p_i \) suspects \( p_j \)”. Otherwise, \( p_i \) regards \( p_j \) as an alive neighbor. Here, we note that before \( p_i \) detects the crash of \( p_j \), \( p_i \) regards \( p_j \) as alive.

Thus, our perfect failure detectors are defined by the following properties:

- Completeness: Eventually, every crashed process is suspected by its neighbors.
- Accuracy: No process is suspected by its neighbors before it crashes.

3. Non-round-based protocol

3.1. Underlying principle

The underlying principle of our protocol is shown in Fig. 2. In the protocol, every process maintains an integer identifier (ID), i.e., the ID of \( p_i \) is \( i \). Without losing generality, we assume that all processes are arranged in a ring with ascending IDs in a clockwise direction (Fig. 1).

Initially, each process creates and sends two messages to collect votes from other processes. A message sent in a clockwise direction is called a RIGHT message and the second message sent in a counterclockwise direction is called a LEFT message. Each process \( p_i \) maintains a local global data item called \( GD^i \). A copy of this local global data item carried by the message created by \( p_i \) is referred to as \( GD^i \). \( GD^i \) is a vector of size \( n \) and contains the values proposed by all processes. Both \( GD^i \) and \( GD^p \) are initialized to \( \{ \bot, \ldots, \bot \} \).

When a message created by process \( p_i \) arrives at process \( p_j \), information is exchanged between \( GD^j \) and \( GD^p \) as follows:

\[
\text{For each } l (0 \leq l \leq n - 1) \text{ if } GD^j[l] \neq \bot \text{ then}
\]

\[
\text{let } GD^j[l] = GD^p[l]
\]

After that, process \( p_i \) forwards the message in the same direction. Each process \( p_i \) maintains two pointers to its closest alive processes, called \( \text{leftalive}_i \) and \( \text{rightalive}_i \). \( \text{Process leftalive}_i \) stores the ID of an alive process, in the counterclockwise direction, closest to \( p_i \), \( p_i \) sends or resends LEFT messages to its \( \text{leftalive}_i \). Conversely, \( \text{rightalive}_i \), in the clockwise direction, RIGHT messages are sent to its \( \text{rightalive}_i \).

Both pointers may change when other processes crash. For example, in Fig. 3, initially, the \( \text{rightalive}_i \) of \( p_i \) is set to \( p_2 \) and the \( \text{leftalive}_2 \) of \( p_3 \) is set to \( p_2 \). After detecting the crash of \( p_2 \), \( p_3 \) sets \( \text{rightalive}_3 \) to \( p_1 \). \( p_2 \) sends \( \text{leftalive}_3 \) to \( p_1 \). After process \( p_1 \) detects the change of \( \text{leftalive}_3 \), it resends all previous messages to \( \text{newleftalive}_3 \), and vice-versa for \( \text{rightalive}_i \).

When a message returns to its creator, it must have traversed all correct processes and traversal stops. When a process detects that the two messages it created have returned, it decides on its local \( GD^i \), sends its decision to all alive neighbors, and terminates the protocol.

In our approach, a message traverses all non-crashed processes one at a time in the chordal ring. When the next process to be traversed has crashed, the message skips the crashed process and traverses a process immediately next to the crashed process according to the direction of the message. If there is a direct connection between the current process and the destination
process, the message is forwarded directly. Otherwise, node-disjoint paths are required to forward the message from the current process to the destination process in the reverse direction, i.e., a LEFT message is normally sent in a counterclockwise direction, but in a clockwise direction for its node disjoint paths.

Fig. 3 shows a reverse traversal in \( C_{16} \langle 4 \rangle \). When \( p_1 \) detects that \( p_2 \) has crashed, it resends all RIGHT messages to \( p_3 \), including those sent from \( p_1 \) to \( p_2 \) previously. However, there is no direct connection between \( p_1 \) and \( p_3 \). In this case, a RIGHT message \( m \) is sent/forwarded/resent via \( 2k + 1 \) node-disjoint paths. Fig. 3 shows three node-disjoint paths \((p_1, p_0, p_{15}, p_3)\), \((p_1, p_{13}, p_9, p_7, p_3)\) and \((p_1, p_5, p_4, p_3)\). When the RIGHT message \( m \) is sent/forwarded/resent via these node-disjoint paths, it is sent/forwarded/resent as a LEFT message. A message \( m \) being sent/forwarded/resent to traverse \( p_3 \) via node-disjoint paths is called a reverse traversal, and message \( m \) is termed a reverse message. Otherwise, it is called a regular message. The process \( p_1 \) is the source of the reverse message and is referred to as a reverse source. The crashing of \( p_2 \) is called a reverse trigger. When \( p_3 \) receives the message \( m \), it ends the reverse traversal and starts forwarding the RIGHT message in a clockwise direction.

3.2. Data structures and definitions

Before introducing the protocol, we introduce the following data structures:

- \( \text{suspect}_i \): a set containing the IDs of \( p_i \)'s neighbors which are suspected to have crashed by the perfect failure detector module of \( p_i \).
- \( \text{new:suspect}_i \): a superset of \( \text{suspect}_i \) consisting of the IDs of crashed neighbors and additional crashed processes detected by the gossip-style failure detection service.
- \( \text{message}(i, GD^p, direction, j, reverse, disjoint-path) \): a message created by \( p_i \) and forwarded from \( p_i \), where \( \text{direction} \) RIGHT (constant variable as 1) denotes a RIGHT message and LEFT (constant variable as 0) denotes a LEFT message. Sometimes, we simply use \( \text{message}(i, GD^p, direction, ...) \) or \( \text{message}(i, GD^p, ...) \) regardless of the sender or the direction. For a regular message, the integer value of \( \text{reverse} \) is set to \(-1\). Otherwise, \( \text{reverse} \) is set to the ID of the reverse source. \( \text{Disjoint-path} \) is a queue of processes, and if it is a regular message, it is set to null. Otherwise, \( \text{disjoint-path} \) maintains the IDs of a node-disjoint path from the reverse source to the destination node initially. Upon the message reverse traverses a process, an ID is dequeued from \( \text{disjoint-path} \).
- \( \text{messagelist}_i \): a list containing all messages sent or forwarded by \( p_i \) except reverse messages. It is used to recover messages missed by a neighbor’s crash. A message \( m \) on leaving the creator node is \( \text{lost} \) if it cannot be resent by any process on its route back to the creator. To reduce memory cost, each item of \( \text{messagelist} \) only contains the message creator ID and direction, \((k, direction)\).
- \( \text{decide}(i, GD) \): a message sent to its neighbors by \( p_i \) as soon as it decides, where \( GD \) is the decision.
3.3. Details of proposed protocol

Each process \( p_i \) invokes the function \( \text{GDC}(v_i) \). It terminates with the invocation of the statement \( \text{return} \) that provides \( \text{Global Data} \). The function as shown in Fig. 4 consists of three concurrent tasks: T1 (lines 3–16), T2 (lines 18–21), T3 (lines 23–27).

- The main task, Task T1, initializes the algorithm, and handles regular and reverse messages. When \( p_i \) detects two returning messages, it decides and sends the decision to all alive neighbors (lines 12–16).
- Task T2 is associated with the processing of a \( \text{decide}(k, \text{GD}) \) message. \( p_i \) multicasts the message to all alive neighbors and decides on \( \text{GD} \) (lines 20–21).
- Task T3 handles process crashes. There are two cases where process \( p_i \) detects the crash of \( p_j \). In the first case, the perfect failure module in \( p_i \) detects the crash of \( p_j \) (in this case, \( p_j \) must be a neighbor of \( p_i \)). In the second case, \( p_i \) receives a crash message from the gossip-style failure detection service indicating that \( p_j \) has crashed.

Before detailing the protocol, we introduce four rules to reduce redundant messages without compromising correctness:

**Rule 1.** If process \( p_i \) receives a message, \( \text{message}(k, \text{GD}_i, \text{direction}, j, \ldots) \), and sender \( p_j \in \text{new} \text{\suspect}(p_i) \), it discards the message and does nothing else.

**Rule 2.** If process \( p_i \) receives a message \( \text{message}(k, \text{GD}_i, \text{direction}, j, \ldots) \), and \( (k, \text{direction}) \in \text{messagelist}(p_i) \), it discards the message and does nothing else.

**Rule 3.** If process \( p_i \) receives a message \( \text{message}(k, \text{GD}_i, \text{direction}, j, g, \text{dis-path}) \), and the destination process (at the tail of \text{dis-path}) is in \text{new} \text{\suspect}(p_i), it discards the message and does nothing else.

**Rule 4.** If process \( p_i \) receives a message \( \text{message}(k, \text{GD}_i, \text{direction}, j, g, \text{dis-path}) \), and the reverse source \( p_g \in \text{new} \text{\suspect}(p_i) \), it discards the message and does nothing else.

The main functions in the protocol – initialization, handling regular and reverse messages, and handling new crashes – are discussed in the following sub-sections.

3.3.1. Initialization

The Initialization function is illustrated in Fig. 4, lines 29–40. Process \( p_i \) initializes its data structures, including preparing node-disjoint paths [25] (lines 31), and creates and sends out two messages (lines 32–38). When sending out a message, \( p_i \) calls another function, \( \text{Forward\_Message}(\ldots) \), as shown in Fig. 4, lines 41–59. In \( \text{Forward\_Message}(\ldots) \), \( p_i \) handles the message according to its direction. If there is a direct connection to the next process (such as \( \text{leftalive}_i \), when the message is a LEFT message), \( p_i \) sends a regular message, \( \text{message}(k, \text{GD}_i, \text{direction}, i, \ldots) \), to its direct neighbors (lines 45–50). Likewise, \( p_i \) sends a reverse message, \( \text{message}(k, \text{GD}_i, \text{direction}, i, \ldots) \), to its alive neighbors (lines 57–58). For each alive neighbor, a different node-disjoint path is attached to the message.

3.3.2. Handling regular messages

In the proposed protocol, we assume every process handles the messages received in First-Come-First-Serve order. The processing of regular messages is illustrated in Fig. 4, lines 60–72. First, we apply rule 1 (lines 62–63) and rule 2 (lines 64–65) to discard useless and redundant messages. Subsequently, \( p_i \) exchanges information with the received regular message (line 66). \( p_i \) calls \( \text{Forward\_Message}(\ldots) \) to forward the message in the same direction of the message if \( p_i \) is not the creator of the message (line 71). Before forwarding the message, \( p_i \) saves the message in the \( \text{messagelist}(\ldots) \), for resending in the future (line 70). If \( p_i \) detects that there are processes missing from \( p_1 \) to \( p_i \), the missing processes must have crashed and \( p_i \) calls \( \text{handle\_crash}(\ldots) \) to handle each crash (lines 67–68). For example, in Fig. 3, if \( p_1 \) detects that \( p_2 \) and \( p_4 \) have crashed, it sends a RIGHT message to \( p_5 \) directly. When \( p_5 \) receives the RIGHT message \( \langle k, \text{GD}_i, \text{RIGHT}, j, 1, \ldots \rangle \), \( p_5 \) can conclude that \( p_2 \) and \( p_4 \) have crashed and calls \( \text{handle\_crash}(\ldots) \) to handle each crash.

3.3.3. Handling reverse messages

How reverse messages are handled is illustrated in Fig. 4, lines 73–91. First, we apply rules 1–4 to discard useless and redundant messages (lines 76–79).

If \( p_i \) is the destination process \( p_h \), we treat the reverse message as a regular message (lines 81–86), but we regard all processes skipped from \( p_h \) according to the regular message sending direction as crashed processes and call \( \text{handle\_crash}(\ldots) \) to handle each crash (lines 82–83). For example, in Fig. 3, if \( p_1 \) knows that both \( p_2 \) and \( p_3 \) have crashed, it will send reverse messages to \( p_4 \) via node-disjoint paths. When \( p_4 \) receives a reverse message \( \langle k, \text{GD}_i, \text{RIGHT}, j, 1, \ldots \rangle \), \( p_4 \) concludes that both \( p_2 \) and \( p_3 \) have crashed.

Otherwise, \( p_i \) forwards the reverse message (lines 88–90) along the node-disjoint path. During this period, no message information exchange occurs, and nothing is added to the \( \text{messagelist}(\ldots) \). For example, in Fig. 3, where \( i = 13 \), \( p_{13} \) receives a \( \text{RIGHT} \) message \( \langle k, \text{GD}_i, \text{RIGHT}, j, 1, \ldots \rangle \). \( p_{13} \) then removes itself from the queue and forwards the reverse message to \( p_h \). Finally, when \( p_h \) receives the reverse message from \( p_i \), it completes the reverse traversal.

3.3.4. Failure detection mechanism

In our proposed protocol, every process has a perfect failure detector to detect the crash of its neighbors. When a process detects a crash, it multicasts the crash information to all its other neighbors. Fig. 5 shows crash information dissemination in \( C_{16}(4) \). When \( p_1 \) detects that \( p_2 \) has crashed, \( p_1 \) multicasts the crash information to its neighbors, \( p_0, p_1, \) and \( p_5 \). \( p_0 \) multicasts the crash information of \( p_2 \) to its neighbors, \( p_{15}, p_{13}, p_{12}, p_4 \), and \( p_5 \) multicasts the crash information to their neighbors.

Our crash failures handling function is illustrated in Fig. 4, lines 92–106. When \( p_i \) detects a new crash or receives a \( \text{crash}(k) \) message, it invokes function \( \text{handle\_crash}(\ldots) \). If \( p_i \) has already known the crash of \( p_j \), it does nothing (lines 94–95). Otherwise, \( p_i \) adds \( p_j \) to \( \text{new} \text{\suspect}(p_i) \), and multicasts the message to all alive neighbors (lines 96–97). If the crashed process causes the change of \( \text{rightalive}_i \) or \( \text{leftalive}_i \), \( p_i \) resends all corresponding regular messages sent previously (lines 98–105).

4. Correctness proof

This section presents the correctness proof to show that our protocol achieves the four desired properties of the \( \text{GDC} \) problem.
Fig. 4. Proposed non-round-based GDC protocol.

1. Function GDC(vi)
2. cobegin
3. task T1:
4. call Initialization(vi);
5. decided ← false;
6. while (not decided)
7. wait until receive a message m;
8. if m is regular message(k, GD^p_k, direction, j, −l, null) then
9. call regular_message(k, GD^p_k, direction, j, −l, null);
10. if m is reverse message(k, GD^p_k, direction, j, g, dis-path) then
11. call reverse_message(k, GD^p_k, direction, j, g, dis-path);
12. if both LEFT and RIGHT messages are returned then
13. decided ← true;
14. end while
15. ∀p ∈ {P, −newsuspect} do send decide(i, GD^p) to p, enddo;
16. return(GD^p);
17. task T2:
18. wait until receive decide(k, GD);
19. ∀p ∈ {P, −newsuspect} do send decide(k, GD) to p, enddo;
20. return(GD);
21. task T3:
22. loop
23. wait until (detect a neighbor, p’s crash) or (receive crash(p));
24. call handle_crash(p);
25. endloop
26. coend
27. Function Initialization(vi)
28. begin
29. prepare node-disjoint paths between p_i and (2k + 2) processes in
30. clockwise side, and another (2k + 2) in counterclockwise side;^2
31. GD^p ← {i, → v_1, ..., l};
32. GD^p ← GD^p;
33. newsuspect ← suspect;
34. rightalive ← closest process in clockwise ∈ newsuspect;
35. leftalive ← closest process in counterclockwise ∈ newsuspect;
36. Forward_Message(i, GD^p, LEFT);
37. Forward_Message(i, GD^p, RIGHT);
38. message_list ← {{i, RIGHT}, (l, LEFT)};
39. end
40. Function Forward_Message(k, GD^p, direction) // Forwarding a Message
41. begin
42. if (direction = RIGHT) then
43. if (p has a direct connection to rightalive_i) then
44. send message(k, GD^p, RIGHT, i, −l, null) to rightalive_i;
45. else
46. ∀p ∈ {P, −newsuspect} do
47. dis-path = a node-disjoint path from p, to rightalive_i
48. including p;
49. send message(k, GD^p, RIGHT, i, i, dis-path) to p;
50. enddo;
51. else
52. if (p has a direct connection to leftalive_i) then
53. send message(k, GD^p, LEFT, i, −l, null) to leftalive_i;
54. else
55. ∀p ∈ {P, −newsuspect} do
56. dis-path = a node-disjoint path from p, to leftalive_i
57. including p;
58. send message(k, GD^p, LEFT, i, i, dis-path) to p;
59. enddo;
60. end

^2 To save memory, p_i is not included in node-disjoint paths called dis-paths. So, a dis-path of p_i only includes IDs from the neighbors of p_i to a destination process.
4.1. Validity property

**Theorem 1.** No spurious initial value. \( \forall i: \text{if } p_i \text{ decides on } GD_i \text{ then } (\forall j: GD_i[j] \in \{v_j, \perp\}). \)

**Proof.** It is obvious that the proposed protocol achieves the validity property. There are two cases for a process to decide. First, a process \( p_i \) determines that its own two messages have returned (Fig. 4, lines 12–13), and then \( p_i \) decides \( GD_i \) (Fig. 4, lines 15–16). In this case, for each \( GD_i[j] \), it is either \( v_j \) or \( \perp \), and this follows directly from the initialization, the exchanging method and the channel reliability (no message alteration, no spurious message). Second, the decision, \( GD_i \), is derived from a received message, \( \text{decide} (k, GD) \) (Fig. 4, lines 19–21). However, initially \( \text{decide} (k, GD) \) is created and sent by process \( p_k \) as in the first case where \( GD_i = GD_k \), and for each \( GD_k[j] \), it is either \( v_j \) or \( \perp \). Thus, the validity property is also achieved in the second case. \( \Box \)

4.2. Termination property

The termination property is guaranteed by Theorem 2. First, we introduce two lemmas.

**Lemma 1.** No message is lost if the message creator does not crash.

**Proof.** A message is lost if it cannot be resent by any process before it returns to its creator. Assume the contrary that a message \( m \) is lost and its creator \( p_i \) does not crash. Without losing generality, we assume message \( m \) is a LEFT message. According to the proposed protocol, there are two cases for a process to send/forward/resend a message. The first case is when \( p_i \) creates \( m \) and sends \( m \) to \( leftalive_i \) (Fig. 4, lines 32–37), or a process \( p_j \) receives \( m \) and
forwards $m$ to $\text{leftalive}_i$ (Fig.4, lines 8–9). The second case is when a process $p_i$ which has sent/forwarded/resent $m$ detects the change of $\text{leftalive}_i$, and then resends $m$ to the new $\text{leftalive}_i$ (Fig.4, lines 103–105).

As $m$ is lost and $m$ does not return to $p_i$, we consider the last non-crashed process $p_k$ closest to $p_i$ in the clockwise direction, which has sent/forwarded/resent $m$ previously. $p_i$ must exist, the worst case being $p_k$. Here, alive processes can be divided into two sets. Set $A$ includes all alive processes that have ever sent/forwarded/resent $m$, and set $B$ includes all alive processes that have not received $m$ including $p_i$. By the contrary hypothesis, no alive process can resend $m$ -- this means every alive process in set $B$ never receives $m$ and every alive process in set $A$ never detects a change of its $\text{leftalive}$. As the change of $\text{leftalive}$ is caused by the crashes of some processes (lines 97–99), $\text{leftalive}_k$ must be alive and be in set $B$. However, according to the system model, alive process $\text{leftalive}_k$ will receive the message eventually (lines 7–9), whether there is a direct connection between $p_k$ and $\text{leftalive}_k$ (lines 52–53) or not (lines 55–58) — a contradiction. 

**Lemma 2.** There exists at least one process which eventually receives its own two messages, and both messages have traversed all non-crashed processes.

**Proof.** Assume the contrary that no process eventually receives its own two messages. As at most $2k + 1 (0 \leq k < (n - 1)/2)$ processes can crash in the protocol, there exists at least one correct process, assuming $p_i$ is correct. First, we consider the LEFT message of $p_i$, $m$. After $m$ is created, it starts to traverse other processes in the chordal ring. By Lemma 1, $m$ will not be lost and will return to $p_i$ eventually, and similarly for the RIGHT message of $p_i$ — a contradiction. Thus, there exists at least one process which eventually receives its own two messages.

Now, assume the contrary that there exists a non-crashed process $p_i$ that message $m$ does not visit when it returns to $p_i$. Based on our perfect failure detectors discussed in Section 2, a process can detect the crash of its neighbors. After a crash is detected, the crash information is disseminated to all other processes (lines 96–97, and lines 25–26). To improve the speed of crash information dissemination, when a process receives a regular message and finds the message has skipped some processes, the process can conclude that all processes skipped have already crashed (lines 67–68 and lines 82–83). No process that is alive can be skipped in this case because the message receiver is the $\text{leftalive}$ or $\text{rightalive}$ of the message sender (lines 98–99). However, by the contrary hypothesis, the non-crashed process $p_j$ must have been skipped — a contradiction.

The same applies to the RIGHT message. Thus both messages returned to the creator have traversed all other non-crashed processes. 

**Theorem 2.** Every correct process decides eventually.

**Proof.** By Lemma 2, we can assume $p_i$ receives its own two messages and send $\text{decide}(i, GD)$ messages to all alive neighbors. There are two cases:

Case 1. $p_i$ is a correct process. According to the protocol, every other correct process can eventually make a decision by receiving either its own two messages (lines 12–16) or $\text{decide}(i, GD)$ messages from other processes (lines 19–21).

Case 2. $p_i$ is not a correct process, and it crashes after sending some $\text{decide}$ messages. There are three sub-cases:

Case 2a. Some correct process receives the $\text{decide}$ message. Thus, every other correct process can eventually make a decision as in case 1.

Case 2b. Some non-crashed process receives the $\text{decide}$ message, and then it takes the same steps as $p_i$ does in Case 2.

Case 2c. No process receives the $\text{decide}$ message. By Lemma 2, another alive process receives its own two messages and takes the same steps as $p_i$.

As there is at least one correct process, case 2b and case 2c will finally result in case 2a or case 1. Thus, all correct processes can decide eventually.

4.3 Agreement property

Theorem 3 shows that the proposed protocol ensures the agreement property. The underlying idea is that when the two messages created by a process return, all non-crashed processes maintain the same $GD$. Thus, the agreement property is ensured. The proof proceeds as follows: Lemma 3 proves that when the two messages created by a process $p_i$ meet at another process $p_k$, process $p_k$ gets complete votes. Next, Lemma 4 proves that when the two messages created by a process return to their creator, referred to as home in the rest of this paper, the $GD$ of all non-crashed processes are equal.

**Lemma 3.** When the two messages created by a process $p_i$ meet at another process $p_k$, process $p_k$ gets complete votes.

**Proof.** Let $m_i^L$ denote the left message created by $p_i$, and $m_i^R$ denote the right message created by $p_i$. Fig. 6 shows the two routes when $m_i^L$ and $m_i^R$ meet at $p_k$: $m_i^L$ traverses processes from $p_i$ to $p_k$ in the counterclockwise direction, $\text{Route}^L_i = (p_i, \ldots, p_k)$ where each process in $\text{Route}^L_i$ has been traversed by $m_i^L$; $m_i^R$ traverses processes from $p_k$ to $p_i$ in the clockwise direction, $\text{Route}^R_i = (p_k, \ldots, p_i)$ where each process in $\text{Route}^R_i$ has been traversed by $m_i^R$.

Assume the contrary that there exist processes (crashed or non-crashed) in $\text{Route}^L_i$ or $\text{Route}^R_i$ that contain a different value $v_h$ when $m_i^L$ and $m_i^R$ meet at $p_k$ but $GD^L_i[h] \neq v_h$. By the contrary hypothesis, these processes do not contain $v_h$ when $m_i^L$ or $m_i^R$ traverses them, and $p_k$ must have crashed before $m_i^L$ or $m_i^R$ tries to traverse $p_h$, which implies that $p_h$ is not in $\text{Route}^L_i$ or $\text{Route}^R_i$; otherwise $GD^L_i[h] = v_h$ — a contradiction.

Consider the set of processes, $\Phi = \{ p \in \text{Route}^L_i \cup \text{Route}^R_i \mid GD^L_i[h] = v_h \}$, when $m_i^L$ and $m_i^R$ meet at $p_k$. There must be a process in $\Phi$, called original process, which causes other processes in $\Phi$ to receive a message containing $v_h$. There may exist several original processes in $\Phi$. However, for each original process $p_i$, there must exist a message $m$ (regular or reverse) containing $v_h$ which
traverses from \( p_1 \) to \( p_j \) after \( m^R \) or \( m^L \) has traversed \( p_j \), where \( p_j \) must be \( \text{leftalive} \) or \( \text{rightalive} \). There are two ways for the message to traverse to \( p_j \) in the regular way and in the reverse way. As \( p_j \) is an original process containing \( v_h \) in \( \Phi \), if \( m \) traverses to \( p_j \) in the regular way, both \( m^R \) and \( m^L \) must not traverse \( p_1 \) — the message sender; if \( m \) traverses to \( p_j \) in the reverse way, both \( m^R \) and \( m^L \) must not traverse \( p_1 \) — the reverse source of \( m \). Without losing generality, we assume \( p_j \subset \text{Route}^L \). There are five cases based on the position between \( p_i \) and \( p_j \).

Case 1. As shown in Fig. 7, \( p_i \) is within the range of \( p_k \) to \( p_j \) in the clockwise direction, where \( p_k \) is the process closest to \( p_j \) in \( \text{Route}^L \) within the range of \( p_i \) to \( p_j \). This means \( m^L \) has traversed from \( p_k \) to \( p_j \) and skipped \( p_i \) (Fig. 4, lines 67–68 or lines 82–83). Then, after \( m^L \) has traversed \( p_i \), \( p_i \) has added \( p_i \) to \( \text{newsuspect} \) (Fig. 4, line 96). By Rule 1, \( p_j \) discards the message \( m \) sent from \( p_k \) if \( m \) is a regular message (Fig. 4, lines 62–63), or by Rule 4, \( p_j \) discards the message \( m \) if \( m \) is a reverse message and the reverse source is \( p_i \) (Fig. 4, lines 76–77). \( p_j \) cannot obtain \( v_h \) in this case—a contradiction.

Case 2. As shown in Fig. 8, \( p_i \) is within the range of \( p_k \) to \( p_j \) in the clockwise direction, where \( p_k \) is the process closest to \( p_j \) in \( \text{Route}^L \) within the range of \( p_i \) to \( p_j \). As \( m^L \) does not traverse \( p_i \) when \( p_k \) arrives \( p_k \), \( p_i \) must have crashed. Here, before \( p_i \) crashes, \( \text{rightalive} \) cannot be set to \( p_i \) (Fig. 4, line 98). Thus, \( p_j \) cannot receive any regular or reverse message from \( p_i \), \( p_j \) cannot obtain \( v_h \) in this case—a contradiction.

Case 3. As shown in Fig. 9, \( p_i \) is within the range of \( p_k \) to \( p_j \) in the clockwise direction where \( p_k \) is the process closest to \( p_j \) in \( \text{Route}^L \) within the range of \( p_i \) to \( p_j \). This means \( m^L \) has traversed from \( p_k \) to \( p_j \) and skipped \( p_i \). Here, \( \text{rightalive} \) is \( p_j \). According to the calculation of \( \text{rightalive} \) (Fig. 4, line 98), \( p_j \) must have been in \( \text{newsuspect} \). By Rule 1, \( p_j \) discards the message \( m \) sent from \( p_k \) if \( m \) is a regular message (Fig. 4, lines 62–63), or by Rule 4, \( p_j \) discards the message \( m \) if \( m \) is a reverse message and the reverse source is \( p_i \). \( p_j \) cannot obtain \( v_h \) in this case—a contradiction.

Case 4. As shown in Fig. 10, \( p_i \) is within the range of \( p_k \) to \( p_j \) in the clockwise direction where \( p_k \) is the process closest to \( p_j \) in \( \text{Route}^L \) within the range of \( p_i \) to \( p_j \). It is clear that \( m^R \) has skipped \( p_i \) when \( m^R \) traverses from \( p_k \) to \( p_j \). If there is a message \( m \) maintaining \( v_h \) traversing from \( p_i \) to \( p_j \), the message must be a \( \text{LEFT} \) message. However, \( \text{leftalive} \) cannot be set to \( p_j \) (Fig. 4, line 99). Thus, \( p_j \) cannot receive any regular message from \( p_i \) or any reverse message with the reverse source as \( p_i \). \( p_j \) cannot obtain \( v_h \) in this case—a contradiction.

Case 5. As shown in Fig. 11, \( p_i \) is within the range of \( p_k \) to \( p_j \) in the counterclockwise direction. The message traversing from \( p_i \) cannot be a \( \text{LEFT} \) message, because the \( \text{LEFT} \) message must first traverse \( p_k \). So, the message must be a \( \text{RIGHT} \) message, and \( \text{rightalive} \) must be set to \( p_i \). All processes within the range of \( p_i \) to \( p_j \) in the clockwise direction, excluding \( p_i \) and \( p_j \), must have crashed (Fig. 4, line 98), i.e., \( p_i \) crashed before \( p_i \) sent the message to \( p_j \). Thus, \( m^L \) must have traversed \( p_i \); otherwise, \( m^L \) will be lost—a contradiction.

Thus, by definition of complete votes, \( p_i \) gets the complete votes when the two messages created by \( p_i \) meet at it.

Lemma 4. If the two messages created by a process return home, the \( \text{GDP} \) of all non-crashed processes are equal.

Proof. Assume the contrary that when the two messages \( i \) (\( \text{RIGHT} \)) and \( message(i) \) (\( \text{LEFT} \)) return, there exists a non-crashed process \( p_k \), where \( \text{GDP}^L(j) \neq \text{GDP}^L(j) \). There are two cases:

The first case is \( \text{GDP}^L(j) = v_j \) but \( \text{GDP}^L(j) \neq v_j \). By Lemma 2, both messages of \( p_i \) must have traversed \( p_k \). This indicates the two messages have met at \( p_k \). When the two messages meet at \( p_k \), \( \text{GDP}^L(j) \neq v_j \) must be true at that time. By Lemma 3, no other process can achieve \( \text{GDP}^L(j) = v_j \) after that time—a contradiction.

Another case is when \( \text{GDP}^L(j) \neq v_j \) but \( \text{GDP}^L(j) = v_j \). By Lemma 2, the two messages must have met at \( p_k \). When the two messages meet at \( p_k \), if \( \text{GDP}^L(j) \neq v_j \) is true at that time, by Lemma 3, no process can achieve \( \text{GDP}^L(j) = v_j \) after that time—a contradiction.
Otherwise, \( GD_i[j] = v_j \) is true at that time, which contradicts \( GD_i[j] \neq v_j \) when the two messages return.

Thus, \( GD_i \) of all non-crashed processes are equal when the two messages created by \( p_i \) return home. 

\textbf{Theorem 3.} No two processes decide differently.

\textbf{Proof.} There are two cases for a process to decide. In the first case, its own two messages return and the process decides and multicasts the decision to its neighbors. In the second case, the process receives a \textit{decide} \( (k, GD_i) \) and then makes the same decision, but \textit{decide} \( (k, GD) \) has been initially created by process \( p_i \). Assume \( p_i \) is the first process whose two messages have returned, makes a decision, and then multicasts the decision to its neighbors. By Lemma 4, every non-crashed process contains the same votes in its \( GD_i \) when \( p_i \) makes its decision. In both of the above cases, when another process makes a decision, it must make the same decision as \( p_i \). Thus, no two processes decide differently.

\section{5. Analysis}

This section focuses on the message complexity of our proposed protocol, and compares our approach with two round-based virtual links approaches. We also consider time complexity at the end of this section.

\subsection{5.1. Message complexity}

\subsubsection{5.1.1. Without failures}

For a system with \( n \) processes, and where initially each process creates and sends two messages, the total number of messages is \( 2n \). When these two messages return home, each message has traversed a total of \( n \) hops. Without considering the \textit{reliable multicast} of \textit{decide} messages, the total number of message hops in the system is \( 2n^2 \). \textit{Reliable multicast} refers to a process multicasting a \textit{decide} message to all neighbors, and each neighbor forwards the received \textit{decide} message to all its neighbors and so on. Considering reliable multicast, the number of resent \textit{decide} messages is less than \( (2k + 2)n \) because each process has \( (2k + 2) \) neighbors and only multicasts once then decides and stops.

The message complexity of \textit{reliable multicast} is:

\[ (2k + 2)n. \]

Thus, message complexity without failures is:

\[ 2n^2 + (2k + 2)n = 2(n + k + 1)n. \]

\subsection{5.1.2. With failures but without reverse traversal}

We consider a message \( m \) traversing from process \( p_i \) to \( p_j \) via \( p_h \). First, we consider a situation without reverse traversal of messages, i.e., \( p_i \) has direct connection to \( p_j \). There are four cases as shown in Fig. 12.

1. In the first case of no failures as shown in Fig. 12(a), \( p_i \) does not crash, message \( m \) is sent from \( p_i \) to \( p_h \) and forwarded from \( p_h \) to \( p_j \). Two message hops are required.
2. In Fig. 12(b), when \( m \) arrives at \( p_i \), \( p_i \) suspects \( p_h \), \( p_i \) forwards \( m \) to \( p_j \) directly. One message hop is saved compared to the first case.
3. In Fig. 12(c), when \( m \) arrives at \( p_i \), \( p_i \) forwards \( m \) to \( p_h \), but \( p_h \) crashes and does not forward \( m \) to \( p_j \). \( p_i \) suspects \( p_h \) and resends \( m \) to \( p_j \). The number of message hops is the same as in the first case.
4. In Fig. 12(d), when \( m \) arrives at \( p_i \), \( p_i \) forwards \( m \) to \( p_h \), and \( p_h \) forwards \( m \) to \( p_j \) before \( p_h \) crashes. When \( p_j \) suspects \( p_h \) has crashed, it resends \( m \) to \( p_j \). One more message hop is needed compared to the first case. According to the protocol, one message is discarded by \( p_j \).

Thus, when \( f \) processes actually crash, the message complexity without considering \textit{reliable multicast} is bounded by \([2n^2 - 2nf, 2n^2 + 2nf]\).

Message complexity plus \textit{reliable multicast} in this case is:

\[ 2n^2 + 2fn + (2k + 2)n = 2(n + f + k + 1)n. \]

\subsection{5.1.3. With failures and reverse traversals}

We consider the case when a reverse traversal of messages is required, i.e., \( p_i \) does not have direct connection to \( p_j \). It is a challenge to determine the topological parameters of chordal rings, such as diameter, bisection width, etc. Even for chordal rings with a single skip link (degree 4), known as double-loop networks, the problem is non-trivial in general, and it is not yet completely solved. However, given a dedicated chordal ring, we can obtain the required topological parameters. The main issue is not in constructing node-disjoint paths for any dedicated chordal ring, but to obtain the message complexity of node-disjoint paths for regular chordal rings.

\textbf{Theorem 5.} For each pair of processes without direct connection in chordal rings, the total length upper bound of all node-disjoint paths is \( n + 2k \), and the lower bound is \( 2(2k + 2) \).

\textbf{Proof.} The lower bound is obviously true. There are \( (2k + 2) \) node-disjoint paths between the two processes, and the length of each node-disjoint path is at least 2 because the two processes have no direct connection.

For the upper bound, we consider constructing \((2k + 2)\) node-disjoint paths between the process pair, \( p_i \) and \( p_j \). Assume all other processes appear in node-disjoint paths. It takes \( n - 2 \) edges to connect all those processes from \( p_i \) and \( 2k + 2 \) edges to connect to the last process \( p_j \). Thus, the upper bound is \( n + 2k \).
For each pair of processes, the total path length is determined by summing all node-disjoint paths. Let $\Delta$ denote the maximum total path length among all pairs of processes. By Theorem 5:

$$2(2k + 2) \leq \Delta \leq n + 2k.$$  
(4)

As there are $f$ actual crashes, and we assume in the worst case each crash causes a reverse traversal of every regular message, there will be at most $2nf$ reverse traversals and each reverse traversal incurs at most $\Delta$ message relays. Thus, the total number of messages without considering reverse traversals is bounded by:

$$[2n^2 - 2nf, 2n^2 + 2nf] + 2nf \Delta.$$  
(5)

Message complexity plus reliable multicast in this case is:

$$2n^2 + 2fn + 2nf \Delta + (2k + 2)n = 2(n + (\Delta + 1)f + k + 1)n.$$  
(6)

The above does not include the messages lost due to process crashes. For example, $p_i$ sends $m$ created by itself to $p_j$, and then both processes crash before $m$ can be forwarded by $p_j$. In this case, $m$ is lost.

5.2. Message complexity comparison

We compare our proposed non-round-based protocol with two round-based virtual links protocols in [8,12]. In the analysis, we do not consider the message overhead introduced by perfect failure detectors.

The protocol in [12] is round-based and needs at most $2f + 2$ rounds. The lower bound is reduced to $f + 2$ rounds in [8]. In every round of both protocols, every process sends a message to another alive process. Thus, in each round, the total number of messages is $n(n - 1)$. As each pair of processes requires $(2k + 2)$ node-disjoint paths to construct a virtual link and incurs at most $\Delta$ overhead messages, the total number of overhead messages is bounded by $n(n - 1)\Delta$ per round. Without failures, both virtual links methods can end within two rounds, requiring $2(n - 1)\Delta$ relay messages. With failures, in the worst case, $(2f + 2)n(n - 1)\Delta$ relay messages are needed in the virtual links method in [12] and $(f + 2)n(n - 1)\Delta$ relay messages are needed in the virtual links method in [8]. Table 1 summarizes the overhead message complexities. As shown, our non-round-based protocol reduces message overhead significantly, and the message complexity in our protocol does not grow linearly with the increase in the number of actual crashes.

5.3. Impact of chordal rings on message complexity

This section discusses the impact of chords on message complexity in our proposed protocol. Message complexity in our proposed protocol depends on $f$ and $\Delta$. However, in our proposed protocol, only reverse traversals incur message overhead.

According to the protocol, for each process $p_i$, initially, leftalive$_i$ equals $p_{(i-1)mod n}$, and rightalive$_i$ equals $p_{(i+1)mod n}$. When $p_{(i+1)mod n}$ or $p_{(i-1)mod n}$ crashes, leftalive$_i$ changes to $p_{(i-2)mod n}$ or rightalive$_i$ changes to $p_{(i+2)mod n}$ respectively. If $p_i$ has no direct connection to the new leftalive$_i$ or rightalive$_i$, reverse traversal occurs. Thus, chordal rings that have consecutive nearby neighbors serve to reduce message complexity further.

We first consider a chordal ring $C$ whose chord $d_1$ equals 2. As the chordal rings considered in this paper are symmetric, each process $p_i$ in $C$ has at least four neighbors, $p_{(i+1)mod n}$, $p_{(i+2)mod n}$, $p_{(i-1)mod n}$ and $p_{(i-2)mod n}$. When $p_{(i+1)mod n}$ or $p_{(i-1)mod n}$ crashes, leftalive$_i$ changes to $p_{(i-2)mod n}$ or rightalive$_i$ changes to $p_{(i+2)mod n}$ respectively. $p_i$ has direct connection to the new leftalive$_i$ or rightalive$_i$, so reverse traversal does not occur in this case. Thus, if no consecutive processes crash, no reverse traversal occurs.

Now, we consider $(2k + 2)$ connected chordal rings $C^0_n$ (2, $\ldots$, $d_k$, $\ldots$, $d_1$) where $2 \leq \alpha \leq k$, and $d_1 = 1 + 1$ when $1 \leq \alpha - 1$ but $d_\alpha > \alpha + 1$.

For $C^n_2$, any two consecutive processes crashing causes reverse traversal. In the worst case, there are $f$ failures, and every message reverse traverses every two consecutive crashes. Thus, the message complexity of reverse traversals is $2nf(2)\Delta$.

For $C^n_3$, any three consecutive processes crashes causing a reverse traversal. In the worst case, there are $f$ failures, and every message reverse traverses every three consecutive crashes. Thus, the message complexity of reverse traversals is $2nf(3)\Delta$.

Our result shows that the proposed GDC protocol is most message efficient for $C^3_n$ or $(2k+2)$-chordal ring $C_n^0$. For $C_n^{(k+1)}$, any $(k + 1)$ consecutive processes crashes causing a reverse traversal. As the chordal ring tolerates at most $(2k + 1)$ crashes, there is at most one reverse trigger in $C_n^{(k+1)}$. Thus, in the worst case, the message complexity of reverse traversals is $2nf(1/k+1)\Delta$. As $f < 2k + 2$, $(2f/(k+1))\Delta < 4n\Delta$.

5.4. Time complexity

The proposed GDC protocol is designed for chordal rings for non-fully connected networks. In time comparison, it appears that our proposed protocol is worse than the current GDC protocol [12]. For fuller comparison, we apply the protocol in [12] to chordal rings using virtual links. First, we consider the case where no crash occurs. Without failures, in our protocol, each message is relayed $n$ times. In [12], two communication steps are incurred per round where in each communication step the longest node-disjoint path without failure is $n/(2k + 2)$, and it needs two rounds to terminate. Thus, the time complexity of our protocol is $n$ compared with $2n/(k+1)$ in [12]. If $k$ becomes smaller and $f$ becomes bigger where $f < 2k + 2$, our protocol is more acceptable in time efficiency.

Second, considering failures, our proposed protocol requires at most $(f + 1)$ reverse traversals and the time complexity is $n + (f + 1)n/(2k + 2)$. However, the protocol in [12] needs $(2f + 2)$
rounds with a time complexity of \( (2f + 2)n/(2k + 2) \). Thus, the time complexity of our protocol is \( n \) compared with \( (f + 1)n/(2k + 2) \) in \([12]\). When \( f \) is bigger, our protocol is more acceptable in time efficiency.

However, the protocol in \([12]\) has to use all node-disjoint paths. This need not be the case in our proposed protocol, and some slow and congested connections can be avoided in the execution of our proposed protocol.

6. Conclusion and future work

In this paper, we have discussed the GDC problem for asynchronous chordal rings. First, we have shown that with virtual links among each pair of processes, the GDC problem in asynchronous chordal rings can be solved by traditional GDC protocols designed for fully connected networks. However, this approach incurs high message overhead. Second, we have addressed the message overhead issue by introducing a new message efficient non-round-based protocol. This protocol addresses the GDC problem and tolerates up to \( 2k + 1 \) crash failures in chordal rings, \( C_{f}([d_1, d_2, \ldots, d_k]) \), with perfect failure detectors. Analysis and comparison show that our protocol reduces message complexity significantly.

In our future work, we shall extend the non-round-based protocol to tolerate other failure models such as the crash-recovery model, omission failure model or malicious failure model. We shall adapt the proposed protocol with weaker failure detectors, and the design of non-round-based GDC protocols for other topologies such as hypercube and multi-mesh.

Acknowledgment

This work is partially supported by the National University of Singapore under Academic Research Fund R-252-000-180-112.

References