Global Data Computation in a Dedicated Chordal Ring

Xianbing WANG[1] and **Yong Meng TEO**[1,2]

[1] Singapore-MIT Alliance, National University of Singapore
[2] Department of Computer Science, National University of Singapore

Abstract* — *Existing Global Data Computation (GDC) protocols for asynchronous systems are designed for fully connected networks. In this paper, we discuss GDC in a dedicated asynchronous chordal ring, a type of un-fully connected networks. The virtual links approach, which constructs t+1 (t<n) process-disjoint paths for each pair of processes without direct connection to tolerate failures (where t is the maximum number of processes that may crash and n is the total number of processes), can be applied to solve the GDC problem in the chordal but the virtual links approach incurs high message complexity. To reduce the high communication cost, we propose a non round-based GDC protocol for the asynchronous chordal ring with perfect failure detectors. The main advantage of our approach is that there is no notion of round, processes only send messages via direct connections and the implementation of failure detectors does not require process-disjoint paths. Analysis and comparison with the virtual links approach shows that our protocol reduces the message complexity significantly.

Index Terms — **data computation, chordal rings, perfect failure detector.**

I. Introduction

In a distributed computation, a Global Data (*GD*) is a vector with one entry being filled with an appropriate value proposed by the corresponding process. The problem of computing a global data and providing each process with a copy, defines the *Global Data Computing* (*GDC*) problem [4, 8]. The problem can be more precisely defined as: In a distributed system with n processes, let $GD[0..n-1]$ be a vector data with one entry per process and let v_i denotes the value provided by p_i to fill its entry of the global data. The *GDC* problem consists of building *GD* and providing a copy of it to each process. Let GD_i denotes the local variable of p_i to contain the local copy of *GD*. The problem is formally specified by a set of four properties as following. Let $\perp$ be a default value that will be used instead of the value v_j when the corresponding process p_j fails prematurely. These properties are [8]:

- *Termination*: Eventually, every correct process p_i decides a local vector GD_i.
- *Validity*: No spurious initial value. $\forall i$: if p_i decides GD_i then $(\forall j: GD_i[j] \in \{v_j, \perp\})$.
- *Agreement*: No two processes decide different Global Data. $\forall i,j$: if p_i decides GD_i and p_j decides GD_j then $(\forall k: GD_i[k] = GD_j[k])$.
- *Obligation*: If a process decides, its initial value belongs to the Global Data. $\forall i$: if p_i decides GD_i then $(GD_i[i] = v_i)$.

According to the well-known FLP consensus impossibility result [6], the *GDC* problem has no deterministic solution in an asynchronous distributed system with process crash failures. To circumvent the impossibility problem, perfect failure detectors defined by Chandra and Toueg [3] is used in a *GDC* protocol to tolerate crash failures (when a process crashes, it stops prematurely and remains crashed forever) for asynchronous distributed systems [8]. The *GDC* protocol proposed in [8] terminates in $\min(2f + 2, t + 1)$ asynchronous computation rounds where f is the number of processes actually crash in an execution. The *GDC* protocol is extended to improve the lower bound to $\min(f + 2, t + 1, n)$ in [4]. However, both *GDC* protocols are based on an assumption that there is a communication channel connecting each pair of processes and channels are reliable. This means the distributed systems are fully connected.

In this paper, we show the first step to solve the *GDC* problem in a type of un-fully connected networks: chordal rings. Chordal rings introduce link redundancy to improve the reliability of ring networks. With alternate paths between processes, the network can sustain several processes and links failures. A chordal ring $C_n \langle d_1, d_2, \ldots, d_k \rangle$ of size n and a chord structure $\langle d_1, d_2, \ldots, d_k \rangle$ is a ring R_n consisting of n processors $\{p_0, p_1, \ldots, p_{n-1}\}$, in which each processor is connected to other processors at *distance* (calculated in clockwise direction) $d_1, d_2, \ldots, d_k, n - d_k, \ldots, n - d_2, n - d_1$ by additional incident chords [14]. Figure 1 shows a chordal ring, $C_{16} \langle 4 \rangle$. Rings and complete graphs are chordal rings, denoted as $C_n \langle\ \rangle$ and $C_n \langle 2, 3, \ldots, \lfloor n / 2 \rfloor \rangle$, respectively.

Existing GDC protocols designed for fully connected systems can be used to solve the GDC problem in un-fully connected systems by adopting a virtual links solution to achieve reliable communication despite possible faults in

X.B. Wang is with the Department of Computer Science, School of Computing, 3 Science Drive 2, National University of Singapore, Singapore 117543, and Singapore-MIT Alliance, 4 Engineering Drive 3, National University of Singapore, Singapore 117576, and is on leave from Computer center, School of Computer, Wuhan University, China 430072 (e-mail: wangxb@comp.nus.edu.sg).

Y.M. Teo is with the Department of Computer Science, School of Computing, 3 Science Drive 2, National University of Singapore, Singapore 117543, and Singapore-MIT Alliance, 4 Engineering Drive 3, National University of Singapore, Singapore 117576 (e-mail: teoym@comp.nus.edu.sg).

any pair of processes without direct connection. The virtual links approach was initially proposed to solve *consensus* problem (a *consensus* problem is for a group of processes to achieve agreement in distributed systems with failures, a *GDC* problem can be regarded as a special *consensus* problem in which processes are required to agree on a global data.) for un-fully connected networks [12, 18], in which *k*-connected networks are considered. $G(V, E)$ represents a undirected graph G with its vertex set V and edge set E, where an edge $e_{ij} \in E$ is a unordered pair $(v_i\ v_j)$, and $v_i, v_j \in V$. A vertex cut of $G(V, E)$ is a subset $V_c \subset V$, such that $G - V_c$ has more than one component. G is said to be a k-connected graph if $\min\ |\ V_c\ | = k$. According to Menger's theorem, G is k-connected iff there exists k disjoint paths between any two vertices of G, and there is no intermediate vertex on these k disjoint paths which will appear twice [18]. The virtual links solution is to establish a virtual link for each pair of processes by constructing $t + 1$ disjoint paths between the two processes, which is a method of achieving reliable communication despite possible faults in relaying processes [12], then the consensus problem in the k-connected topology can be solved by existing consensus protocols designed for fully connected systems. How to construct disjoint paths to establish virtual links can be found in [16].

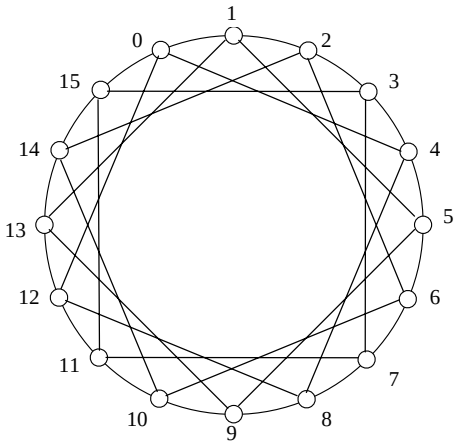

Figure 1. $C_{16}\langle 4\rangle$ Chordal Ring

An un-fully connected chordal ring, $C_n\langle d_1, d_2, \ldots, d_k\rangle$ with $2k$ different chords, is a $(2k + 2)$-connected networks, because $C_n\langle d_1, d_2, \ldots, d_k\rangle$ is still a connected graph after deleting any $2k + 1$ nodes. The dedicated chordal ring considered in this paper is $C_n\langle 2, \ldots, \lfloor (t + 1) / 2 \rfloor\rangle$ which is a $(2\lfloor (t + 1) / 2 \rfloor)$-connected networks, where t $(2 < t < n - 1)$ is the number of failures that can be tolerated. If t is 1, the considered chordal ring is $C_n\langle\ \rangle$, and when t is 2, the considered chordal ring is $C_n\langle 2\rangle$. In this model, when $t \geq n - 2$ and n is even, $C_n\langle 2, \ldots, \lfloor (t + 1) / 2 \rfloor\rangle$ is fully connected; when $t \geq n - 3$ and n is odd, $C_n\langle 2, \ldots, \lfloor (t + 1) / 2 \rfloor\rangle$ is fully connected. We can establish virtual links for each pair of processes in the dedicated un-fully connected chordal ring, then the *GDC* problem in the dedicated chordal ring can be solved by the round-based *GDC* protocols in [8, 4].

The shortcoming of the virtual links approach is the high message complexity incurred. In each round, the number of messages increases significantly because it relies on the use of disjoint-paths. In the asynchronous systems, another important issue is how to implement failure detectors. There are two methods: 1) use virtual links for each pair of processes to detect each other's crash by adopting traditional implementation scheme [3]; 2) use gossip-style failure detection service [15], after a process detect one neighbor's crash, it multicasts the crash information to all its neighbors. It is obvious that both methods incur many redundant messages.

To reduce the message overhead, we propose a non round-based *GDC* protocol for the dedicated asynchronous chordal ring that has no notion of round, processes only send messages via direct connections and the implementation of failure detectors does not require process-disjoint paths. Non round-based *GDC* protocol is first introduced in [17] for fully connected asynchronous distributed systems with perfect failure detectors. In this paper, we propose a protocol where every process just needs to detect its neighbor's failures and is not required to multicast failures. Analysis and comparison with virtual links solution shows that our protocol reduces the message complexity significantly.

The rest of this paper is organized as follows. Section II introduces the system model. Section III describes our proposed non-round *GDC* protocol for the dedicated asynchronous chordal ring. In section IV we present the correctness proof of the proposed protocol. Section V analyzes the message complexity and compares it with the virtual links solution. Section VI concludes the paper with a discussion of future works.

II. System Model

An asynchronous chordal ring $C_n\langle 2, \ldots, \lfloor (t + 1) / 2 \rfloor\rangle$ consist of n processes tolerating at most t failures, $\prod = \{p_0, \ldots, p_{n-1}\}$, where $2 < t < n - 1$. When t is 1, the chordal ring is $C_n\langle\ \rangle$, and when t is 2, the chordal ring is $C_n\langle 2\rangle$. There is no shared memory in the system and processes exchange messages by ring edges and chords. Channels are reliable, i.e., no spurious messages, no loss and no corruption but need not be FIFO. Moreover, process speeds and communication delays are arbitrary. For any pair of processes p_i and p_j, they are neighbors of each other if there is a ring edge or a chord connecting p_i and p_j. We denote the set of all neighbors of p_i as $\prod_i$.

The failure model considered is the crash failure model. When a process crashes, it definitely stops its activity and does nothing else [11]. In the asynchronous chordal rings, each process p_i is equipped with a failure detector module. This module provides p_i with a set of variables called $suspected_i$ that contains the identities of the neighbors that is assumed to have crashed. Process p_i can only read this

set of variables, which is continuously updated by the module. If $p_j \in suspected_i$, we say that "p_i suspects p_j". Otherwise, p_i regards p_j as an *alive neighbor*. According to the quality of guesses made by failure detector modules, eight classes of failure detectors can be defined [3]. In this paper, we use *perfect failure detector* [2, 8], in which no guess is mistaken. It is defined by the following properties:

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

How to realize the perfect failure detector is mentioned in [8].

III. The Non Round Based Protocol

A. Underlying Principle

The underlying principle of the protocol is shown in Figure 2. Initially, each process creates and sends two messages to collect votes from other processes. One message is sent in the clockwise direction, called RIGHT message, and another is sent in the anti-clockwise direction, called LEFT message. Each process resends received messages in the same direction after exchanging information with these messages (each process maintains a local *GD*, and every message carries a copy of the local *GD'*, when a message arrives on a process, the process combines the two *GD*s. The pseudocode for the exchange progress is: **For each** i $(0 \le i \le n-1)$ **if** $GD'[i] \neq \perp$ **then** let $GD[i] = GD'[i]$ **endif**; **let** $GD' = GD$;). When one message returns to its creator, the message must have traversed all correct processes. When a process detects that the two messages created by itself return, it decides its local *GD* and sends its decision to all alive neighbors, then terminates the algorithm.

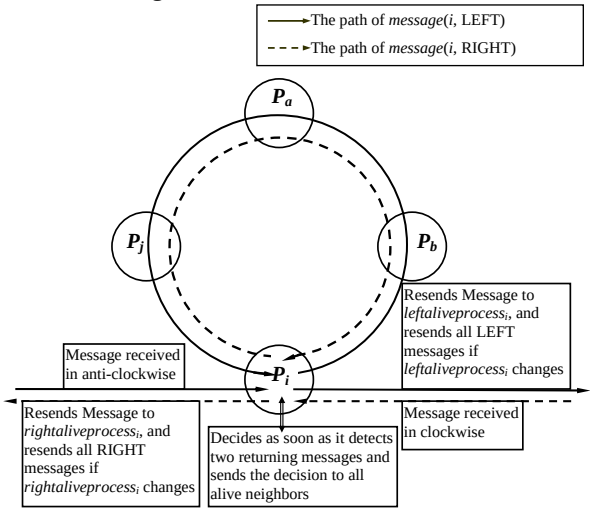

Figure 2. Proposed Non Round-based Protocol

In our approach, a message needs to traverse other processes one by one in the ring. When $\lfloor (t+1)/2 \rfloor$ or more than $\lfloor (t+1)/2 \rfloor$ consecutive processes have crashed, some alive processes may have been skipped when the message is resent if the chordal ring is not fully connected. In this case, to traverse these skipped processes, a *reverse traversal* is needed for the message. Figure 3 shows the reverse traversal, in which each process only has four neighbors. p_i detects that p_{i+1} and p_{i+2} crashed, p_i sends a RIGHT message m to neighbor p_{i-2}. In this case, m skips many correct processes. m will be resent in the clockwise direction in order to traverse all skipped correct processes. Message m being resent to traverse p_j, the alive process closest to p_i, is called a *reverse traversal*. In a reverse traversal, message m is called a *reverse message*. Otherwise, it is called a *regular message*. The process p_i is the *reverse causer*, and those alive processes been skipped are in the *reverse range*. The crash of these consecutive processes, such as crash of p_{i+1} and p_{i+2} in the above example is called a *reverse trigger*. To reduce the message overhead and save time, when p_{i-2} resends m in reverse direction after exchanging information with m, it sends m to an alive neighbor, p_{i-4}, in reverse direction, which is close enough to p_i. When p_j detects there is no alive neighbor between p_j and p_i in the reverse range, it ends the reverse traversal and starts resending the regular message.

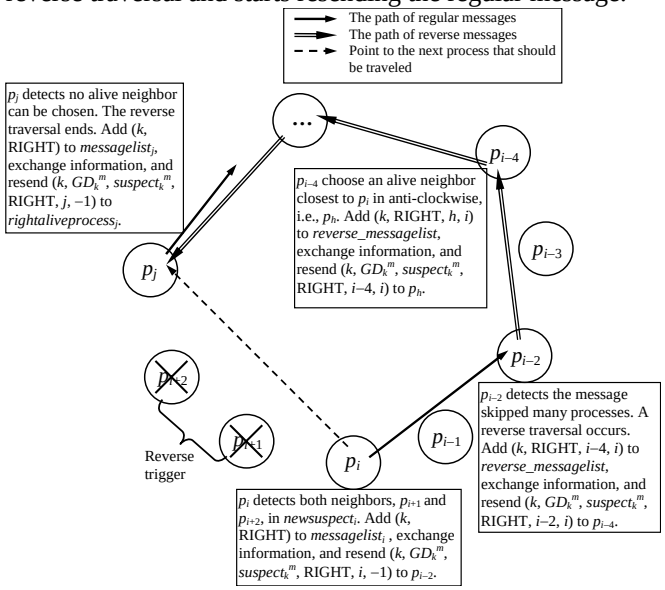

Figure 3. Reverse Message Traversal

B. Data Structures and Definitions

Before introducing the protocol and its correctness proof, the following data structures are introduced.

- *ID*, every process maintains an integer number as its identity, i.e., the *ID* of p_i is i. Without losing generality, in this paper we assume all processes are arranged in a ring with ascending *ID*s in clockwise direction (Figure 1). Thus, $p_{(i+1) \bmod n}$ is at clockwise to p_i.
- GD_i^p, the local *GD* maintained by process p_i. It is an n-size vector which contains the values proposed by processes and exchanges information with each visiting message; $GD_i^p[j]$ is used to contain the proposed value of the process p_j. Initially, GD_i^p contains only v_i, as $\{\perp, \ldots, v_i, \ldots, \perp\}$.

- GD_i^m, the copy of the local *GD* maintained by a message created by p_i, *message*(i, GD_i^m). It is an n-size vector which contains the same information as GD_i^p does. When the message traverses process p_j, exchange GD_i^m and GD_j^p (**For each** k ($0 \le k \le n-1$) **if** $GD_i^m[k] \neq \perp$ **then** let $GD_j^p[k] = GD_i^m[k]$ **endif**; **let** $GD_i^m = GD_j^p$;). Initially, GD_i^m is equal to GD_i^p.
- $suspect_i$, a set containing *ID* of its neighbors which are suspected to have crashed by the perfect failure detector module of p_i.
- $newsuspect_i$, a superset of $suspect_i$, it also contains *ID* of other processes (not only its neighbors) which are deduced to have crashed from the receiving messages. Because correct processes may not suspect their common neighbor's crash at the same time, this can be used to save crash detecting time.
- $suspect_i^m$, similar to $newsuspect_i$, but it carried by a message and includes all crashed processes. When a message created by p_i traverses process p_j, p_j exchanges $suspect_i^m$ and $newsuspect_j$ (**Let** $suspect_i^m = suspect_i^m \cup newsuspect_j$; **Let** $newsuspect_j = suspect_i^m$). To reduce the message and memory size, $suspect_i$, $newsuspect_i$ and $suspect_i^m$, can be implemented as an n-bit vector, jth bit related to the failure state of process p_j, 0 means not crash, 1 crashed.
- $leftaliveprocess_i$, contains *ID* of one of p_i's neighbors which is alive and closest to p_i in p_i's anti-clockwise direction, process p_i sends or resends LEFT messages to its $leftaliveprocess_i$. It is always the *ID* of the alive neighbor with the biggest distance to p_i.
- $rightaliveprocess_i$, similar to $leftaliveprocess_i$ but in clockwise direction.
- *message*(i, GD_i^m, $newsuspect_i^m$, *direction*, j, *reverse*), a message created by p_i and sent from p_j where *direction*, RIGHT (constant variable as 1) is a RIGH message or LEFT (constant variable as 0). Sometimes, we just use *message*(i, GD_i^m, *direction*) or *message*(i, GD_i^m) regardless of the sender or the direction. *reverse* is an integer, when the message is a regular message, *reverse* equals to −1. Otherwise, *reverse* equals to the ID of the process which has caused the reverse traversal.
- $messagelist_i$, a list containing all messages sent or resent by p_i except the reverse messages. It is used to recover messages missed by a neighbor's crash. A message m is *lost* if it cannot be resent by any process anymore. To reduce the memory cost, each item of $messagelist_i$ only contains the message's creator and direction, (k, *direction*).
- $reverse\_messagelist_i$, a list containing all reverse messages sent or resent by p_i. For recovering reverse messages missed by a neighbor's crash. Each item contains (k, *direction*, g, j), where j is the ID of the reverse causer, and g is ID of the destination process.
- *decide*(i, *GD*), is a message sent to its neighbors by p_i as soon as it decides, where *GD* is the decision.

Definition 1. *Traversed and Reverse Traversed. We say a message has traversed a process p_i if message(k, GD_k^m, , direction, , –1) has visited p_i and (k, direction) stored in $messagelist_i$. We say a message has reverse traversed a process p_i if message(k, GD_k^m, , direction, , g) or message(k, GD_k^m, , direction, g, –1) has visited p_i and (k, direction, ,g) stored in $reversemessagelist_i$.*

Definition 2. *Complete votes. A non-crashed process p_i collects complete votes if all other non-crashed processes do not maintain a different value v_j, i.e., if $GD_i^p[j] \neq v_j$, there is no other non-crashed process, such as p_k, with $GD_k^p[j] = v_j$.*

Definition 3. *Meet. The two regular messages created by one process meet at process p, if one type message finds that another message with the same creator in the messagelist of p.*

C. Proposed Protocol

Each process p_i invokes the function GDC(v_i). It terminates with the invocation of the statement **return**() that provides the *Global Data*. The function consists of three concurrent tasks: *T*1, *T*2 and *T*3, as shown in Figure 4.

```
1. Function GDC(v_i)
2. cobegin
3. task T1:
4.   call Initialization(v_i);
5.   decided ← false;
6.   while (not decided)
7.     wait until receive a message m;
8.     if m is regular message(k, GD_k^m, direction, j, –1) then
9.       call handle_regular_message(k, GD_k^m, suspect_k^m, direction, j, –1)
10.      if m is reverse message(k, GD_k^m, suspect_k^m, direction, j, g) then
11.        call handle_reverse_message(k, GD_k^m, suspect_k^m, direction, j, g)
12.    if detected two returning messages then
13.      decided ← true
14. end while
15. ∀p_j ∈ (∏_i – newsuspect_i) do send decide(i, GD_i^p) to p_j enddo;
16. return(GD_i^p)
17.
18.task T2:
19. wait until receive decide(k, GD);
20. ∀p_j ∈ (∏_i – newsuspect_i) do send decide(k, GD) to p_j enddo;
21. return(GD)
22.
23.task T3:
24. loop
25.   wait until detect a new neighbor, p_k's crash;
26.   call handle_new_crash(p_k);
27. endloop
28.coend
```

Figure 4. Pseudocode of the Protocol

The main task, Task *T*1, initializes the algorithm, handles regular and reverse messages, and when it detects two returning messages, p_i decides and resends the decision to all alive neighbors. Task *T*2 is associated with the processing of a *decide*(k, *GD*) message. p_i resends the message to all alive neighbors and decides on *GD*. Task *T*3 handles new crashes and recovers missed messages.

Firstly, we introduce two rules to reduce redundant messages and to ensure agreement correctness.

Rule 1. If process p_i receives a message, *message*(k, GD_k^m, $suspect_k^m$, *direction*, j), and finds the sender p_j is suspected by $newsuspect_i$, p_i discards the message and does nothing else.

Rule 2. If process p_i receives a message *message*(k, GD_k^m, $suspect_k^m$, *direction*, j, −1) and finds that (k, *direction*) ∈ $messagelist_i$, or receives a reverse message *message*(k, GD_k^m, $suspect_k^m$, *direction*, j,) but finds (k, *direction*, ,) is already in $reverse\_messagelist_i$, p_i discards the message and does nothing else.

Rule 2 includes: When p_i receives a message *message*(k, GD_k^m, $suspect_k^m$, *direction*, j, −1) and finds it requires reverse traversal, but (k, *direction*, ,) is already in $reverse\_messagelist_i$, p_i discards the message also.

Now we describe the main functions on initialization, how to handle messages (where rule 1 and rule 2 will be applied firstly) and new crashes. The details on these functions are in the Appendix.

- **Initialization(v_i)**. p_i initializes its data structures, then creates and sends out two messages.
- **handle_regular_message(k, GD_k^m, $suspect_k^m$, *direction*, j, −1).** We assume every process treats the receiving messages in FIFO manner (i.e., p_i will exchange information with the receiving messages in FIFO manner). If the message requires reverse traversal, to reduce the message overhead and save time, process p_i will choose a neighbor closest to p_j in the reverse range to send the reverse message. Otherwise, p_i resends the message if p_i is not the creator of the message. If new crashed neighbors found according to the message sender, p_j, (there are common neighbors between p_i and p_j), **call *handle_new_crash*()** to handle new crashes.
- **handle_reverse_message(k, GD_k^m, $suspect_k^m$, *direction*, j, g)**. To reduce the message overhead and save time, p_i chooses a neighbor closest to p_g in the rest reverse range (the rest reverse range includes the processes from p_i to p_g in the reverse range) to resend the message. If no process can be chosen, the reverse traversal ends and continues the message's regular travesal.
- ***handle_new_crash*(p_k).** If the crashed process causes the change of $rightaliveprocess_i$ or $leftaliveprocess_i$, p_i will resend all corresponding regular messages ever sent. If p_i has sent reverse messages to p_k before, p_i will try to choose a new process to resend each reverse message.

IV. Correctness Proof

This section proves that the proposed protocol achieves the four properties of the *GDC* problem.

A. Validity Property

Theorem 1. *If p_i decides GD_i then* $\forall j$: $GD_i[j] \in \{v_j, \perp\}$.

Proof. There are two cases for a process to decide. First, a process p_i finds its own two messages returned, then p_i makes decision as GD_i. According to the protocol, for each $GD_i^p[j]$, it is either v_j or ⊥, it follows directly from the initialization, the exchanging method and the channel reliability (no message alteration, no spurious message). Second, the decision *GD* is derived from a received message, *decide*(k, *GD*). But, initially *decide*(k, *GD*) is created and sent by process p_k in the first case and *GD* is the decision made by p_k, the theorem must be true in this case too. ❑

B. Termination Property

The termination property is guaranteed by Theorem 2. Firstly, we introduce two Lemmas.

Lemma 1. (1) *At most one reverse trigger in an execution of the protocol;* (2) *no further-reverse traversal occurs during a reverse traversal;* (3) *there is reliable communications between any pair of processes in the reverse range.*

Proof. Because the proposed protocol is designed to tolerate up to t crashes but each process has $2\lfloor(t + 1) / 2\rfloor$ neighbors, it is clear that there is at most one reverse trigger in any execution of the protocol. Thus during a reverse traversal, no further-reverse traversal can occur.

For any pair of non-crashed processes, p_i and p_j in the reverse range, if there are less than $\lfloor(t + 1) / 2\rfloor$ processes between them, there will be a ring edge or chord connecting them, p_i can send the reverse message to p_j directly. Otherwise, relay processes are required and each process has at least $\lfloor(t + 1) / 2\rfloor$ neighbors in the reverse range. Because there is at most t failures but the chordal ring is a $(2\lfloor(t + 1) / 2\rfloor)$-connected networks, and a reverse trigger already exists, so only less than $\lfloor(t + 1) / 2\rfloor$ processes may crash. By Menger's theorem, a link can be constructed between p_i and p_j within the reverse range. Thus, there are reliable communications between p_i and p_j in the reverse range. This means any reverse traversal can end and recover the traversal eventually. ❑

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. Because at most t ($t < n$ − 1) processes can crash in the protocol, there exists at least two correct processes, assume one is p_i. Next, we will prove eventually p_i will receive its own two messages.

First, we consider the LEFT message, m, of p_i. After it is created, m starts to traverse other processes in the ring. According to the protocol, m traverses the alive processes in anti-clockwise. If there is an un-traversed alive process,

p_j, eventually m can traverse it as guaranteed by the protocol and Lemma 1.

In the same way, p_i eventually receives its RIGHT message and the message has 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 sends *decide*(i,GD) messages to all other alive neighbors. There are two cases:

CASE 1. p_i is a correct process. Because chordal rings are (t + 1)-connected graph and at most t processes crash, then every other correct process can eventually make decision by either receiving its own two messages or receiving *decide*(i, GD) message from others.

CASE 2. p_i is not a correct process, it crashes after it sending some *decide* messages. There are three sub-cases:

case 2a. some correct processes receive the *decide* message, thus every other correct process can eventually make decision as in CASE 1.

case 2b. some non-crashed processes receive the *decide* message, then it will take the same steps as p_i does in CASE 2.

case 2c. no process receives the *decide* message, by Lemma 2, another process will receive its own two messages and take the same steps as p_i does.

Because 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. ❑

C. Agreement Property

We show that the proposed protocol ensures agreement property by Theorem 3. The underlying idea is that when a process' two messages return, all non-crashed processes maintain the same GD^p, thus, the agreement property is ensured. The proof proceeds as follows: Firstly, Lemma 3 proves when the two messages created by a process p_i meet at another process p_k, the process p_k gets complete votes. Then, Lemma 4 proves that when the two messages created by a process return home, the GD^p of every non-crashed process will be equal.

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

Proof. Let m_i^{L} denotes the left message created by p_i, and m_i^{R} denotes the right message created by p_i. Assume the contrary that there exist processes (crashed or non-crashed) traversed by m_i^{L} or m_i^{R} maintain a different value v_h when m_i^{L} and m_i^{R} meet at p_k but $GD_k^p[h] \neq v_h$. According to the protocol, these processes did not maintain v_h when m_i^{L} or m_i^{R} traversing them, and p_h must have crashed before m_i^{L} or m_i^{R} tried to traverse p_h. Assume p_j is one of the *original processes* traversed by m_i^{L} or m_i^{R} which maintains v_h when m_i^{L} and m_i^{R} meet at p_k. Without losing generality, assume m_i^{R} have traversed p_j. So, there exists another message m traversed p_j which maintained v_h after m_i^{R} traversed p_j and both m_i^{L} and m_i^{R} did not traverse the sender of m, p_l, until they met at p_k. It is obvious p_l is a neighbor of p_j.

Case 1. Consider m as a regular message. If m_i^{R} should have traversed p_l before it traverses p_j, there are two cases: 1). When m_i^{R} received by p_j, p_l was skipped, then p_l was added to p_j's *newsuspect*, by Rule 1, m would be discarded by p_j. 2) There exists another neighbor of p_j, p_g, when m_i^{R} received by p_g before m_i^{R} traverses p_j, p_l was skipped, according to the protocol, no regular message sent from p_l to p_j. Both cases are contrary to p_j maintaining v_h. Otherwise, m_i^{R} should traverse p_l after it traversed p_j, there are also two cases: 1). When m_i^{R} resent by p_j, p_l was skipped, then p_l must be in p_j's *newsuspect*, by Rule 1, m would be discarded by p_j. 2) There exists another neighbor of p_j, p_g, when m_i^{R} resent by p_g after m_i^{R} traversed p_j, p_l was skipped, according to the protocol, no regular message sent from p_l to p_j. Both cases are contrary to p_j maintaining v_h.

Case 2. Consider m as a reverse message. If p_l is not the reverse causer, the proof is the same as in Case 1.

Thus, according to the definition of *complete votes*, p_k gets the *complete votes* when the two messages created by a process p_i meet at it. ❑

Lemma 4. *If the two messages created by a process return home, the GD^p of every non-crashed process will be equal.*

Proof. By Lemma 2, when a message returns home, it must have traversed all non-crashed processes. Now, assume the contrary, that when process p_i's two messages *message*(i, RIGHT) and *message*(i, LEFT) return, there exists a non-crashed process p_k at the same time, $GD_i^p[j] = v_j$ but $GD_k^p[j] \neq v_j$. As the above proof, both messages must have traversed p_k. This indicates the two messages have met on p_k. Because the exchange algorithm between GD^m and GD^p is a union operation, when the two messages met on p_k, $GD_k^p[j] \neq v_j$ must be true at that time. By Lemma 3**,** no other process can achieve $GD^p[j] = v_j$ after that time. This is a contradiction.

Another case is $GD_i^p[j] \neq v_j$ but $GD_k^p[j] = v_j$. For same reason, the two messages must have met on p_k. When the two messages met on p_k, if $GD_k^p[j] \neq v_j$ is true at that time, by Lemma 3, no process can achieve $GD^p[j] = v_j$ after that time. This is a contradiction. Otherwise, $GD_k^p[j] = v_j$ is true at that time, it is contradiction to $GD_i^p[j] \neq v_j$ when the two messages return according to the protocol.

Thus GD^p of all non-crashed processes will be equal after the two messages created by p_i return home. ❑

Theorem 3. *No two processes decide differently.*

Proof. There are two cases for a process to decide. The first case is its own two messages return, then the process decides and multicasts the decision to neighbors. The second case is the process receives a *decide*(*k*, *GD*) then makes the same decision, but *decide*(*k*, *GD*) has been initially created by a process under first case.

Assume p_i is the first process, whose two messages return and decides then multicasts the decision to neighbors. By Lemma 4, every non-crashed process contains the same votes in its GD^p after p_i made decision. In both of the above cases when another process makes decision, it must make the same decision as p_i made. Thus, no two processes decide differently. ❑

D. Obligation Property

Theorem 4. *If a process decides, its initial value belongs to the Global Data.* $\forall i$: *if* p_i *decides* GD_i *then* $(GD_i[i] = v_i)$.

Proof. When a process, p_i, decides after its two messages return, its two messages maintain v_i, then obligation property is satisfied following the initialization. Another case is p_i decides after it received a *decide*(*k*, *GD*) message, the *decide* message must be sent from p_k initially. When p_k makes the decision, its two messages must have returned and have traversed p_i. According to the protocol, v_i must be in *GD*. So in this case obligation property is also ensured. ❑

V. Message Complexity Analysis

First, we consider no failure occurs.

A. Message Complexity without Failures

For a system with *n* processes and initially each process creates and sends two messages, the total number of messages is 2*n*. When the two messages created by a process return home, the total number of hops of each message is *n*. Without considering the *reliable multicast*[1] of *decide* messages, the total number of message hops in the system is $2n^2$. Considering reliable multicast, the number of resent *decide* messages will be less than $2\lfloor(t + 1)/2\rfloor n$, because each process has $2\lfloor(t + 1)/2\rfloor$ neighbors and only multicasts once then decides and stops.

B. Message Complexity with Failures

Now consider failure occurs, but firstly without reverse traversal of messages.

There are four cases for a message *m* traversing from process p_i to p_j via p_h as shown in Figure 5.

[1] Because a *decide* message will send to all neighbors like a multicast, and each neighbor will resend received *decide* message to its all neighbors and so on, then we call the whole process as a *reliable multicast*.

1. In the case of no failure as shown in Figure 5a, p_h does not crash; message *m* will be sent from p_i to p_h and then be sent from p_h to p_j. Two message hops are required.
2. In Figure 5b, when *m* arrives at p_i, p_i suspects p_h, then p_i resends *m* to p_j directly. In this case, one message hop is saved comparing to Figure 5a.
3. In Figure 5c, when *m* arrives at p_i, p_i resends *m to* p_h but p_h crashes and does not resend *m* to p_j. Then p_i suspects p_h and resends *m* to p_j. In this case, the number of message hops is the same as in Figure 5a.
4. In Figure 5d, when *m* arrives at p_i, p_i resends *m to* p_h and p_h resends *m* to p_j before it crashes. Then p_i suspects the crash and resends *m* to p_j. In this case, one more message hop is needed comparing to Figure 5a. According to the protocol, one message will be discarded by p_j.

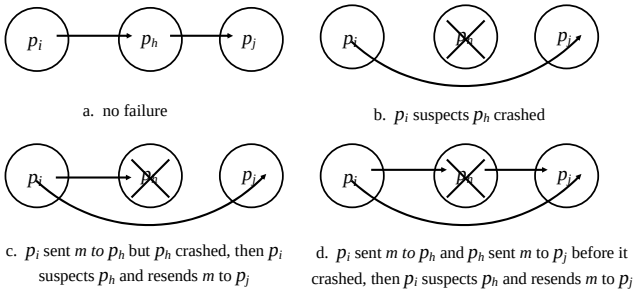

Figure 5. Message routes

Thus, when *f* processes actually crash, the message complexity of the protocol without considering *reliable multicast* is bounded by $[2n^2 - 2nf, 2n^2 + 2nf]$.

Now, consider reverse traversal. First, according to the protocol, only consecutive $\lfloor(t + 1) / 2\rfloor$ neighbors of a process have crashed; the message sent or resent from the process need a reverse traversal. Second, the speed for a message traversing in reverse is much faster than a regular message. By Lemma 1, there is at most one reverse trigger in an execution, thus, the total number of messages is less than $2[2n^2 - 2nf, 2n^2 + 2nf]$.

The previous discussion does not include the message lost by process crash. For example, p_i sends *m* created by itself *to* p_h and p_h resends *m* to p_j, then all the three processes crash before *m* is resent by p_j.

C. Comparing with Other Protocols

We compare the proposed protocol with the protocols in [8, 4] but we do not consider the message overhead caused by implementing failure detectors.

Firstly, we compute the process-disjoint paths between any pair of processes, p_i and p_j, in the dedicated chordal ring. If $p_j \in \prod_i$, there is no process-disjoint path. Otherwise, the length can be bounded by *t* and *n*, i.e., the length of the longest disjoint path is $\lfloor n/\lfloor(t + 1)/2\rfloor\rfloor$, the average length is $\lfloor n/(t + 1)\rfloor$ from p_i to $p_{(i+n/2) \bmod n}$, and the shortest is two obviously. Without losing generality, assume *t* is odd. $t + 1$ process-disjoint paths are needed. Any two process-disjoint paths in different directions are

made of a ring, requiring about $2n/(t + 1)$ relay messages. Thus, a total of about n relay messages is needed for p_i to reliably communicate with p_j in the dedicated chordal ring.

Perfect failure detectors are used to solve *GDC* problem in asynchronous system [8]. The protocol is round-based and needs at most $2f + 2$ rounds. The lower bound is improved to $f + 2$ rounds in [4]. In each round, every process sends a message to another alive process. Thus, in each round, the total number of messages are $n(n - 1)$. When considering process-disjoint paths, total relay messages is $n^2(n - 1)$ per round. It is obvious that our protocol reduces the message complexity significantly for solving *GDC* problem in the dedicated chordal rings. When consider failures, in the worst case, $(2f + 2)n^2(n - 1)$ relay messages are needed in [8] and $(f + 2)n^2(n - 1)$ relay messages are needed in [4]. But, the message complexity of our proposed protocol does not increase linearly when the number of actual crashes increases.

On the time comparison, the time for a process to relay a message in process-disjoint paths is less than the time for a process to handle a receiving message and resending it. So, we only compare the communication time. Without failures, in our protocol, each message is relayed n times. Both protocols in [8, 4] need at least two rounds to stop without failures. In each round, the longest process-disjoint path is $n/(t + 1)$ when no failure occurs. Thus, the time complexity of our protocol and other protocols [8, 4] is n *versus* $2n/(t + 1)$ respectively. If t is small, our protocol is acceptable. With failures, the relay times of our protocol double but is acceptable in comparison, i.e., the time complexity of protocol in [8] and [4] increase by factor of $(2f + 2)$ and $(f + 2)$ respectively.

VI. Conclusion

In this paper, we present the *GDC* protocols for a dedicated asynchronous chordal ring. We show that with virtual links among each pair of processes, the *GDC* problem in the dedicated chordal ring can be solved by the tradition *GDC* protocols which are designed for fully connected networks. But it incurs a high message overhead. To reduce the message complexity, we propose a message-efficient non-round based protocol for solving the *GDC* problem tolerating up to t crash failures in the chordal ring with perfect failure detectors. Analysis and comparison show that our protocol reduces the message complexity significantly.

Future works include (1) investigate protocols to solve the *GDC* problem for regular chordal rings, $C_n \langle d_1, d_2, \ldots, d_k \rangle$; (2) investigate the possibility of extending the non-round based protocol to tolerate other failure models such as crash-recovery model, omission failure and malicious failure model; (3) adapt the proposed protocol with weaker failure detectors; and (4) design of non round-based *GDC* protocols for other topologies such as hypercube and multi-mesh.

References

1. P.A. Bernstein, V. Hadzilacos, and N. Goodman, "*Concurrency Control and Recovery in Database Systems*", Reading, Mass.: Addison-Wesley, 1987.
2. B. Charron-Bost, R. Guerraoui, and A. Schiper, "*Synchronous system and perfect failure detector: solvability and efficiency issues*", Proc. IEEE Int. Conf. on Dependable Systems and Networks (DSN), New York, USA, June 2000, pp. 523-532.
3. T. Chandra and S. Toueg, "*Unreliable Failure Detectors for Reliable Distributed Systems*", J. ACM, vol. 43, no. 2, Mar. 1996, pp. 225-267.
4. C. Delporte-Gallet, H. Fauconnier, J. Hélary, M. Raynal, "*Early Stopping in Global Data Computation*", IEEE Trans. Parallel Distrib. Syst. 14(9), 2003, 909-921.
5. D. Dolev, R. Reischuk, and R. Strong, "*Early Stopping in Byzantine Agreement*", J. ACM, vol. 37, no. 4, Apr. 1990, 720-741.
6. M.J. Fischer, N. Lynch, and M.S. Paterson, "*Impossibility of Distributed Consensus with One Faulty Process*", J. ACM, vol. 32, no. 2, Apr. 1985, 374-382.
7. V.K. Garg and J. Ghosh, "*Repeated Computation of Global Functions in a Distributed Environment*", IEEE Transaction on Parallel and Distributed Systems, vol. 5, no. 8, pp. 823-834, 1994.
8. J. Hélary, M. Hurfin, A. Mostéfaoui, M. Raynal, and F. Tronel, "*Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors*". IEEE Transactions on Parallel and Distributed Systems 11(9): 897-909 (2000)
9. J.M. Helary and M. Raynal, "*Synchronization and Control of Distributed Systems and Programs*", John Wiley & Sons, 1990.
10. L. Lamport, "*Time, Clocks and the Ordering of Events in a Distributed System*", *Communications of the ACM* 21(7), 1978, 558-565.
11. N. Lynch, "*Distributed Algorithms*", Morgan Kaufmann, 1996.
12. F. J. Meyar, and D. K. Pradhan, "*Consensus with Dual Failure Modes*", IEEE Transaction on Parallel and Distributed Systems, Vol. 2, No. 2, April 1991, 214-222.
13. A. Mostéfaoui, and M. Raynal, "*Consensus Based on Failure Detectors with a Perpetual Accuracy Property*", in Proc 14th Int'l Parallel and Distributed Processing Symp. 2000.
14. B. Mans, and N. Santoro, "*Optimal Fault-Tolerant Leader Election in Chordal Rings*", The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing (FTCS 1994), June 15-17, 1994, Austin, Texas.
15. R. Renesse, Y. Minsky, and M. Hayden, "*A Gossip-Style Failure Detection Service*", Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware'98, Sep. 15-18, 1998.
16. D. Sidhu, R. Nair, and S. Abdallah, "*Finding disjoint paths in networks*", ACM SIGCOMM Computer Communication Review, Proceedings of the conference on Communications architecture & protocols, 21(4), Aug. 1991, 43-51.
17. X. Wang, and J. Cao, "*A non Round-based Consensus Protocol in Asynchronous Distributed Systems with Perfect Failure Detectors*", Technical paper, 2003.
18. D. B. West, "*Introduction to Graph Theory*", Prentice Hall, NJ, USA. 1996.
19. S.C. Wang, K.Q. Yan, and C.F. Cheng, "*Asynchronous Consensus protocol for the unreliable un-fully connected network*", Operating Systems Review 37(3), 2003, 43-54.

Appendix

Details of the main functions in the proposed protocol.

Initialization(v_i)

```
1. Function Initialization(v_i)
2. begin
3.    GD_i^p ← {⊥, …, v_i , …, ⊥};
4.    GD_i^m ← GD_i^p;
5.    newsuspect_i ← suspect_i;
6.    rightaliveprocess_i ← closest neighbor in clockwise
          ∉ newsuspect_i;
7.    leftaliveprocess_i ← closest neighbor in anti-
          clockwise  ∉ newsuspect_i;
8.    if (the first consecutive ⌊(t+1)/2⌋ neighbors in
          clockwise crashed) then
9.       rightaliveprocess_i ← leftaliveprocess_i
10.   if (the first consecutive ⌊(t+1)/2⌋ neighbors in
          anticlockwise crashed) then
11.      leftaliveprocess_i ← rightaliveprocess_i
12.   send message(i, GD_i^m, RIGHT, i, −1) to
          rightaliveprocess_i;
13.   send message(i, GD_i^m, LEFT, i, −1) to
          leftaliveprocess_i;
14.   messagelist_i ← { (i, RIGHT), (i, LEFT)};
15.end
```

Figure 6. Initialization

Processing Regular Message

```
1. Function handle_regular_message(k, GD_k^m, direction, j,
−1)
2. begin
3.    if (j ∈ newsuspect_i) then
4.       return;           //applying rule 1
5.    if ((k, direction) ∈ messagelist_i) then
6.       return;           //applying rule 2
7.    Exchange information between GD_i^p and GD_k^m;
8.    if ((k, direction, ) ∈ reverse_messagelist_i) then
9.       delete (k, direction, ) from
          reverse_messagelist_i;
10.   for each common neighbor, p_h, of p_i and p_j do
11.      call handle_new_crash(p_h);
12.   if (k ≠ i) then
13.      add (k, direction) to messagelist_i;
14.      if (the coming consecutive ⌊(t+1)/2⌋ neighbors
             crashed) then
15.         send message(k, GD_k^m, direction, i, i) to
                leftaliveprocess_i or rightaliveprocess_i
                according to the reverse direction
16.      else
17.         send message(k, GD_k^m, direction, i, −1) to
                leftaliveprocess_i or rightaliveprocess_i
                according to the message direction
18.end
```

Figure 7. Processing Regular Message

Processing Reverse Message

```
1. Function handle_reverse_message(k, GD_k^m, direction, j,
g)
2. begin
3.    if (j ∈ newsuspect_i) then
4.       return;        //applying rule 1
5.    if ((k, direction, ) ∈ reverse_messagelist_i) then
6.       return;        //applying rule 2
7.    if ((k,direction) ∈ messagelist_i) and (p_k ∉ rest
          reverse range) then
8.       return;   //the reverse traversal ended already
9.    Exchange information between GD_i^p and GD_k^m;
10.   for each common neighbor, p_h, of p_i and p_j do
11.      call handle_new_crash(p_h);
12.   if not reach the reverse trigger then
13.      add (k, direction, g) to reverse_messagelist_i;
14.      send message(k, GD_k^m, direction, i, g) to
             leftaliveprocess_i or rightaliveprocess_i
             according to the reverse direction
15.      return;
16.   if reach the reverse trigger then
17.      delete (k, direction, ) from
             reverse_messagelist_i;
18.      add (k, direction) to messagelist_i;
19.      if (k ≠ i) then
20.         send message(k, GD_k^m, direction, i, −1) to
                leftaliveprocess_i or rightaliveprocess_i
                according to the message direction
21.end
```

Figure 8. Processing Reverse Message

Processing New Crash

```
1. Function handle_new_crash(p_k)
2. begin
3.    if (k ∈ newsuspect_i) then
4.       return;
5.    newsuspect_i ← newsuspect_i ∪ {k};
6.    re-calculate rightaliveprocess_i and
          leftaliveprocess_i   //Figure 6 line 6-11
7.    if (rightaliveprocess_i has changed) then
8.       For each (j, RIGHT) ∈ messagelist_i do
9.          if (the coming consecutive ⌊(t+1)/2⌋ neighbors
                crashed) then
10.            send message(j, GD_i^p, RIGHT, i, i) to
                   leftaliveprocess_i
11.         else
12.            send message(j, GD_i^p, RIGHT, i, −1) to
                   rightaliveprocess_i
13.      For each ((j, LEFT, g) ∈ reverse_messagelist_i) do
14.         if not reach the reverse trigger then
15.            send message(j, GD_i^p, LEFT, i, g) to
                   rightaliveprocess_i
16.         if reach the reverse trigger then
17.            delete (j, LEFT, g) from
                   reverse_messagelist_i;
18.            add (j, LEFT) to messagelist_i;
19.            if (j ≠ i) then
20.               send message(j, GD_i^p, LEFT, i, −1) to
                      leftaliveprocess_i
21.   if (leftaliveprocess_i has changed) then
22.      For each (j, LEFT) ∈ messagelist_i do
23.         if (the coming consecutive ⌊(t+1)/2⌋ neighbors
                crashed) then
24.            send message(j, GD_i^p, LEFT, i, i) to
                   rightaliveprocess_i
25.         else
26.            send message(j, GD_i^p, LEFT, i, −1) to
                   leftaliveprocess_i
27.      For each ((j, RIGHT, g) ∈ reverse_messagelist_i)
do
28.         if not reach the reverse trigger then
29.            send message(j, GD_i^p, RIGHT, i, g) to
                   lefttaliveprocess_i
30.         if reach the reverse trigger then
31.            delete (j, RIGHT, g) from
                   reverse_messagelist_i;
32.            add (j, RIGHT) to messagelist_i;
33.            if (j ≠ i) then
34.               send message(j, GD_i^p, RIGHT, i, −1) to
                      rightaliveprocess_i
35.end
```

Figure 9. Processing New Crash