Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, IEEE Computer Society Press, Hong Kong, May 10-12, 2004

An Approach to Achieve Message Efficient Early-Stopping Uniform Consensus Protocols

Xianbing Wang[1,2,3], Jiannong Cao[4] and Yong Meng Teo[1,2]

[1]*Department of Computer Science, National University of Singapore, Singapore 117543*
[2]*Singapore-MIT Alliance, Singapore 117576*
[3]*College of Computer Science, Wuhan University, China*
{wangxb, teoym}@comp.nus.edu.sg

[4]*Internet and Mobile Computing Lab*
Department of Computing,
Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
csjcao@comp.polyu.edu.hk

Abstract

Existing consensus protocols for synchronous distributed systems concentrate on the lower bound on the number of rounds required for achieving consensus. This paper proposes an approach to reduce the message complexity of some uniform consensus protocols significantly while achieving the same lower bound in which for any t-resilient consensus protocol only $t + 1$ processes engaging in sending messages in each round.

1. Introduction

Consensus is one of the fundamental problems in distributed computing. Assuming a distributed system with a set of n processes, $\{p_1, p_2, \ldots, p_n\}$, in the Consensus problem, each process p_i initially proposes a value v_i, and all non-faulty processes have to decide on one common value v which is equal to one of the proposed values. A process is *faulty* during an execution if its behavior deviates from that prescribed by its algorithm, otherwise it is *correct*. More precisely, the Consensus problem is defined by the following three properties:

- T*ermination*: Every correct process eventually decides on a value.
- *Validity*: If a process decides on v, then v was proposed by some processes.
- *Agreement*: No two correct processes decide differently.

The agreement property applies to only correct processes. Thus it is possible that a process decide on a distinct value just before crashing. The *Uniform Consensus* prevents such a possibility and replace Agreement as:

- *Uniform Agreement*: No two processes (correct or not) decide differently.

Consensus has been extensively studied over last two decades both in synchronous and asynchronous distributed systems. In a synchronous distributed system, message delays and relative processes' speed are bounded and the bounds are known. In contrast, none of these bounds exist in an asynchronous distributed system. In [7], Fischer, etc. proved that consensus could not be solved deterministically in an asynchronous system that is subject to even a single crash failure. The impossibility result for consensus stems from the inherent difficulty of determining whether a process has actually crashed or is only very slow. One way to circumvent this impossibility is the *unreliable failure detector* concept proposed by Chandra and Toueg [3].

Most consensus protocols that have been designed so far are based on the notion of *round*, such as the protocols proposed in [6, 10, 13] for synchronous systems and those in [3, 8, 11] for asynchronous systems with *unreliable failure detector*s. And these are designed to tolerate crash failures. When a process crashes in a round, it sends a subset of the messages that it intends to send in that round, and does not execute any subsequent rounds [10]. If a protocol allows processes to reach *consensus* in which at most t ($t < n - 1$) processes can crash, the protocol is said to tolerate t faults or be *t-resilient*.

If a protocol can achieve consensus and *stops* before the $t + 1$ round when there are actually f ($f \le t$) faults, we call it an *early-stopping* protocol. In [6], the well-known lower bound, $\min(t + 1, f + 2)$, for early-stopping consensus protocols in synchronous distributed systems has been proved. If just consider the time at which processes decide, we call those protocols in which all processes decide before round $t + 1$ with actual f faults as *early deciding* protocols. It has been proved that the lower bound of early deciding uniform consensus protocols is $f + 2$ rounds, where $f < t - 1$ [1, 9]. For $f = t - 1$, the uniform consensus only require $f + 1$ rounds [1]. The lower bound of early deciding consensus protocols is $f + 1$ rounds [1].

In this paper, we focus on message-efficient consensus protocols for synchronous systems, which still achieve the lower bound, $\min(t + 1, f + 2)$ rounds. The system model is as follows. In a synchronous distributed system, there is a set of n processes, $\prod = \{p_1, \ldots, p_n\}$, that communicate and synchronize with each other by sending and receiving messages. The underlying communication system is

assumed to be failure-free: there is no creation, alteration, loss or duplication of message.

We introduce an approach, which use the *Pigeon Hole* Principle to design a uniform consensus protocol which achieves the lower bound but needs fewer messages than the existing protocol. Initially, $t + 1$ processes, called *Coordinating Process* (*CP*), are randomly chosen. Other processes are called non-*CP*s. Therefore there is at least one *CP*, which never crashes. At the first round, each process sends messages to coordinators. Thereafter, only coordinators will send messages to other processes. Non-*CP*s just receive messages and make decision after they find every *CP* can make decision. We apply this method to an existing early-stopping uniform consensus protocol proposed in [13] which leads to reduction in message complexity. The message complexity is defined as the number of messages sent by all the participating processes in the worst case of the protocol.

Compare to the *Rotating Coordinator Paradigm* [5, 2], another way to reduce the message complexity, in which only the coordinator impose a value as decision in each round, our proposed protocol with simple revision can solve many other applications in distributed systems such as the GDC problem [8, 4].

The rest of this paper is organized as follows: Section 2 presents the early-stopping uniform consensus protocol proposed in [13]. Section 3 presents our message efficient uniform consensus protocol and the correctness proof. Section 4 describes the Rotating Coordinator Paradigm and its restriction. Section 5 extends our message-efficient protocol to solve the GDC problem. Finally, Section 6 concludes this paper.

2. The uniform consensus protocol [13]

Figure 1 presents the protocol, Protocol_RAY02, for synchronous systems proposed in [13]. The protocol is a variant of the well-known *flood-set* consensus protocol [10]. When considering early-stopping or early-deciding, each process must broadcast a message at each round, even if its content is the empty set [13]. In this paper, we consider early-stopping uniform consensus protocols.

Protocol_RAY02 can tolerate up to t ($t < n$) crash failures. Each process p_i has a unique identity (ID) i ($1 \le i \le n$). Each process p_i invokes the function Consensus(v_i) where v_i is the value it proposes. It terminates with the invocation of the **return**() statement that provides it the decided value. Consensus() is made up of $t + 1$ rounds whose aim is to fill in an array $V_i[1..n]$. In such a way, $V_i[j]$ contains the value proposed by p_j. The flood-set strategy used to attain this goal is particularly simple: it consists of a process to send new information it got during the previous round in each round. If a process finds no process crashes in a round, it can decide in the next round.

Protocol_RAY02 can achieve uniform consensus. The proof is straightforward and can be found in [13]. The message complexity of Protocol_RAY02 is $\min(t + 1, f + 2) \times n^2$, because in each round all processes will broadcast messages.

```
Function Consensus(v_i)
V_i ← [⊥,…,v_i,…, ⊥]; New_i ← {(v_i, i)}
r = 0;
while r < t + 1 do
   r = r + 1;
   foreach j: send (New_i) to p_j;
   let R_i(r) = {process from which message have been received during r}
   let rec_from[j] be the set received from p_j during r (be ∅ if no
   message);
   New_i ← ∅
   foreach j ≠ i: foreach (v, k) ∈ rec_from[j]:
      if (V_i[k] = ⊥) then V_i[k] = v; New_i ← New_i ∪ {( v, k)} endif
   if (R_i(r) = R_i(r−1) ∧ r < t + 1) then
      set a flag to direct p_i to decide at the end of r + 1 endif
end while
let v = the first non-⊥ value of V_i;
return (v)
```

Figure 1. The uniform consensus protocol [13]

Protocol_RAY02 only briefly mentioned that a flag is set to direct a process to decide at the end of the next round if it has learned all information that can be known. But it does not mention sending the decision flag or decision to other processes. A full version of Protocol_RAY02 can be found in [12] and flag transmitted by a variable called w_i.

3. The Proposed Uniform Consensus Protocol

The *Pigeon Hole* Principle is used to design the new message efficient uniform consensus protocol. Initially, $t + 1$ processes are randomly chosen as *Coordinating Processes* to achieve uniform consensus using the traditional protocol. Only *Coordinators* can send messages in the protocol after the first round. Other processes just send message in the first round and then wait for receiving messages and make decision after they find every Coordinating Process can make decision.

3.1. The Protocol

In this protocol, every process p_i has a unique identity i. Without losing generality, we choose the first $t + 1$ processes, whose identities range from 0 to t, as *CP*s, $\prod_{CP} = \{CP_i \mid 0 \le i \le t\}$. Other processes are called non-*CP*s, $\prod_{non\text{-}CP} = \prod - \prod_{CP}$.

The protocol is illustrated in Figure 2. It consists of two functions, Consensus() executed by *CP*s (Figure 2a) and Consensus2() executed by non-*CP*s (Figure 2b). Function Consensus() is adopted from Protocol_RAY02. Each CP_i invokes the function Consensus(v_i) where v_i is the value it proposes. It terminates with the invocation of the statement **return**() that provides the decided value. Comparing to Protocol_RAY02, at each round, CP_i not only broadcast a message including New_i but also broadcast a Boolean

variable *done* in the message which indicates if CP_i can make decision at the end of the round or not, *done* is true means CP_i can do so. $R_i(r)$, initiated as $R_i(0) = \prod_{CP} - \{p_i\}$, is a set of *CP*s from which CP_i receives a message during round r. If it is the same as $R_i(r-1)$, to the knowledge of CP_i, there are no *CP* crashes during round r. Then it gets the *full* information on the votes initially proposed by all processes. To ensure the uniform agreement property, it can make decision at the end of next round because at that time every other *CP* may keep the same value set as it keeps.

At the first round both *CP*s and non-*CP*s send messages to *CP*s. So, in Consensus(), another set $R'_i(r)$, initiated as $R'_i(0) = \prod - \{p_i\}$, is used for *CP*s to judge whether it receives messages from all processes at the first round. If no crash occurs during the first round, every *CP* will set *done* to true at the end of the first round. The purpose of letting non-*CP*s to broadcast its value in the first round is to solve some problem such as the GDC problem. If a *CP* finds the *done* of another *CP* is true in a round, it will set its *done* to true also.

```
Function Consensus(v_i)                    % Executed by CPs %
V_i ← [⊥, …, v_i, …, ⊥]; New_i ← {(v_i, i)}; R_i(0) = ∏_CP − {p_i};
R'_i(0) = ∏ − {p_i}; done = false;
r = 0;            % r: round number %
while r < t + 1 do
   r = r + 1;
   foreach j ≠ i: send (New_i, done) to p_j;
   if (r = 1) then
      let R'_i(r) = {processes from which messages have been received
      during r};
      let R_i(r) = {CPs from which messages have been received
      during r};
      let rec_from[j] be the set received from p_j during r (be ∅ if no
      message);
      New_i ← ∅;
      foreach j ≠ i: foreach (v, k) ∈ rec_from[j]:
         if (V_i[k] = ⊥) then V_i[k] = v; New_i ← New_i ∪ {(v, k)} endif;
      if (R'_i(r) = R'_i(r − 1)) then done = true endif;
   else
      let R_i(r) = {CPs from which messages have been received
      during r};
      let rec_from[j] be the set received from CP_j during r (be ∅ if no
      message);
      if (done) then let v = first non-⊥ value of V_i; return(v) endif;
      let rec_done[j] be the Boolean value of done received from CP_j
      during r (be false if no message);
      New_i ← ∅;
      foreach j ≠ i: foreach (v, k) ∈ rec_from[j]:
         if (V_i[k] = ⊥) then V_i[k] = v; New_i ← New_i ∪ {(v, k)} endif;
      if (∪_{1≤j≤t+1} rec_done[j] = true) then done = true; endif;
      if (R_i(r) = R_i(r − 1)) then done = true endif;
   endif
end while
let v = first non-⊥ value of V_i;
return(v)
```

Figure 2a. The protocol executed by *CP*s

In Consensus2(), each non-*CP* maintains a copy vector of each *CP* maintained in the previous round. If it receives a message from a *CP* and the *CP*'s *done* is true, this means if the *CP* does not crash at end of this round, it must make decision. So if it finds all messages received in this round indicating all senders can make decision at the end of the round, the non-*CP* can make decision too. If this case does not occur, at the end of round $t + 1$, the non-*CP* will make decision based on its own vector.

```
Function Consensus2(v_i)                % Executed by non-CPs %
V_i ← [⊥, …, v_i, …, ⊥]; New_i ← {(v_i, i)}; R_i(0) = ∏_CP;
foreach CP_j: V_i,j ← [⊥, …, ⊥, …, ⊥];
r = 0;            % r: round number %
while r < t + 1 do
   r = r + 1;
   if (r = 1) then foreach CP_j: send New_i to CP_j endif;
   let R_i(r) = {CPs from which messages have been received during
   r};
   let rec_from[j] be the set received from CP_j during r (be ∅ if no
   message);
   let rec_done[j] be the Boolean value of done received from CP_j
   during r (be true if no message);
   foreach j: foreach (v, k) ∈ rec_from[j]:
      if (V_i,j[k] = ⊥) then V_i,j[k] = v; endif;
      if (V_i[k] = ⊥) then V_i[k] = v; endif;
   if (R_i(r) = ∅) then R_i(r) = {all CPs in R_i(r−1) which indicates done
   in previous round } endif;
   if (∩_{1≤j≤t+1} rec_done[j] = true) then
      let v = first non-⊥ value of V_i,j where CP_j ∈ R_i(r); return(v)
   endif;
end while
let v = first non-⊥ value of V_i;
return(v)
```

Figure 2b. The protocol executed by non-*CP*s

3.2. Correctness prove

We need to prove the proposed protocol satisfy the three properties of uniform consensus.

Lemma 1. At the end of round r, if a *Coordinating Process* CP_i sets *done* to *true*, then it maintains all information that can be known.

Proof. We need to prove that there does not exist a pair (v, k), $(v, k) \in CP_j$ $(j \neq i)$ and CP_j does not crash but $(v, k) \notin CP_i$. According to the protocol, there are three cases for CP_i to be set *done* to true at the end of round r:

Case 1. $R'_i(r) = R'_i(r-1)$ where $r = 1$, $R'_i(1) = R'_i(0)$. And we have $R'_i(0) = \prod - \{p_i\}$. Then CP_i receives messages from every other process in the round. CP_i will set *done* to true and maintain all information that can be known.

Case 2. $R_i(r) = R_i(r-1)$ where $r > 1$. Assume the contrary, there exists a pair $(v, k) \in CP_j$ $(j \neq i)$ and CP_j does not crash but $(v, k) \notin CP_i$. In this case, $R_i(r) = R_i(r-1)$ at the end of round r $(r > 1)$, CP_i must receive a message from CP_j. Because (v, k) is not included in the message according to the assumption, then CP_j must know (v, k) before $r - 1$ round. Anyway, according to the protocol it will send (v, k) to CP_i before round r. Thus it is contradiction to $(v, k) \notin CP_i$. Therefore CP_i maintains all the information that can be known and set *done* to true by the end of round r. But it can not know if other *CP*s maintain the same information. Hence, CP_i proceeds to

round $r + 1$ (to send the new pairs), decide and terminate at end of round $r + 1$.

Case 3. CP_i finds another CP_j set *done* to true at round $r - 1$. It set *done* to true at the end of round r. Because according to the protocol, it must contain every pair CP_j contains. According to the previous discussion, CP_j maintains all the information that can be known. Thus CP_i maintains all information that can be known. ❑

Lemma 2. If two *CP*s set *done* to *true* at the end of the same round, they maintain the same information.

Proof. It is obviously true following Lemma 1. ❑

Lemma 3. If a *CP*, CP_i, decides at the end of the round r, all other *CP*s that does not decide in round r must maintain the same information as CP_i.

Proof. By Lemma 1, CP_i set *done* to *true* at the end of round $r - 1$, it maintains all information that can be known. According to the protocol, all other *CP*s that does not decide in round r must get all information from CP_i at the end of round r, they will maintain the same information as CP_i and set *done* to *true* at the end of round r. ❑

Lemma 4. If a *CP* decides at the end of the round r, all other processes must make decision by the end of round $r + 1$.

Proof. Suppose CP_i decides at the end of round r.

If $r = t + 1$, according to the protocol, all processes that have not made decision will decide in this round.

Now, consider $r < t + 1$. If there are some *CP* can not make decision in round r, CP_i must set *done* to *true* at the end of round $r - 1$ and inform others in round r, then those *CP*s that does not decide in round r will set *done* to *true*. Then in round $r + 1$, those *CP*s will decide and all non-*CP*s can find all *CP*s can decide and make decision also.

Otherwise, if all other *CP*s make decision in round r or before round r, all non-*CP*s can find all *CP*s can decide and make decision in round r. ❑

Theorem 1. The protocol solves the Uniform Consensus problem in synchronous systems where up to t processes can crash.

Proof. The Validity and Termination properties are obviously true.

Now, we show the uniform agreement property is also achieved. We will show that any two processes maintain the same information before they decide, thus they decide on the same value. There are 3 cases of the relationships of each two processes: the first case is two *CP*s, the second case is a *CP* and a non-*CP*, and the third case is two non-*CPs*.

Case 1 is proved by contradiction, assume the two *CP*s, CP_i and CP_j, decide on different values.

Firstly assume they decide in the same round r.

- If $r < t + 1$, CP_i and CP_j must set *done* to be *true* at the end of round $r - 1$. By Lemma 2, both CP_i and CP_j maintain the same information and can not make different decision, - a contradiction.
- Otherwise, $r = t + 1$.
  - CP_i and CP_j set *done* to be *true* at the end of round $r - 1$. A contradiction can be reached by Lemma 2.
  - Only one of them set *done* to be *true* at the end of round $r - 1$. Without losing generality, assume CP_i does so, by Lemma 3, CP_j will maintain the same information as CP_i, it can not make a different decision, - a contradiction.
  - Both of them do not set *done* to be *true* at the end of round $r - 1$. Because at most t processes can crash, there must exist a round r' ($r' \leq t + 1$), in which no process fails. Then in round r', all *CP*s can set *done* to be *true*. Then, according to the assumption, CP_i and CP_j will set *done* to be *true* at the end of round r. By Lemma 2, both of them maintain the same information, - a contradiction.

Secondly assume the two *CP*s, CP_i and CP_j, decide in different rounds, CP_i decide in round $r1$ and CP_j decides in round $r2$, without losing generality, assume $r1 < r2$. By Lemma 3, when CP_i decides in round $r1$, CP_j will maintain the same information with CP_i, which is the full information. It can not make a different decision, - a contradiction.

The case 1 ensures that each pair of *CP*s makes the same decision and they maintain the same information in their vector when they decide.

Case 2 and **case 3** can be proved that any non-*CP* makes the same decision as a *CP* does. There are three cases for a non-*CP* to make decision.

- The first case is that the non-*CP* finds all non-crashed *CP*s can make decision. Then it make decision based on a copy vector of one non-crashed *CP*, which sent a message in the same round indicating that it decides in that round. By Lemma 2 and 3, in this case the non-*CP* must make the same decision as the *CP* does.
- The second case is that the non-*CP* finds all non-crashed *CP*s can make decision, but it does not receive any message in the round. Then it makes decision based on a copy vector of a *CP*, which sent a message in the previous round indicating deciding in that round, (The *CP* must exist in this case. Otherwise, by Lemma 4, no *CP* can make decision, which is contrary to at most t processes can crash among $t + 1$ *CP*s.). By Lemma 2 and 3, in this case the non-*CP* must make the same decision as the *CP* does also.
- The third case is the non-CP makes decision based on its own vector at the end of round $t + 1$. According to the protocol, there must exist a crash in each of previous t rounds, otherwise at least one round without failures all non-crashed *CP*s can set *done* to *true* before round $t + 1$. Thus, there is no failure in round $t + 1$. As proved previous, every *CP* maintains the full information in this case, and then any pair of *CP*s maintains the same vector. Thus, every non-*CP* must maintain the same information in their vector V_i as a *CP* does.

Thus, the Theorem must be true. ❑

Theorem 2. Every process that decides will do so at the end of a round r, where $r \le \min(t+1, f+2)$.

Proof. It is obvious that all non-*CP*s can make decision and terminate in a round $r \le \min(t+1, f+2)$, if all *CP*s not crashed decide by the end of round r. So we just need prove every coordinating process that decides will do so at a round $r \le \min(t+1, f+2)$. There will be two cases:

Case 1. $\min(t+1, f+2) \ne f+2$, which means $t+1 < f+2$. Then CP_i decides at the end of the round $t+1$ and all other not-crashed processes will decide in this round. The Theorem is true in this case.

Case 2. $\min(t+1, f+2) = f+2$. In the first $f+1$ rounds, there must be a round r in which no crash occur. So, that at least one *CP* set *done* to true by the end of round f or all *CP*s set *done* to true by the end of round $f+1$ must be true. Both let all non-crashed *CP*s decide by the end of round $f+2$. ❑

3.3. Cost of the protocol

According to the protocol, during the first round, every *CP* broadcast a message, and every *non-CP* only send messages to all *CP*s, then the message complexity in this round is: $(t+1) \times (n-1) + (t+1) \times (n-t-1)$. After the first round, only *CP*s broadcast a message during each round, *non-CP*s just receive messages. Then the message complexity in each of those round is: $(t+1) \times (n-1)$. By Theorem 2, the protocol needs at most $\min(t+1, f+2)$ rounds, thus, the message complexity of our proposed protocol is:

$$\min(t+1, f+2) \times (t+1) \times (n-1) + (t+1) \times (n-t-1).$$

Comparing to Protocol_RAY02, during the first round, our protocol send less messages because non-*CP*s just send messages to *CP*s, the number of messages decreases by $(n-t-1)^2$. After that, non-*CP*s just receive messages from *CP*s. Then during each round, the number of messages decreases by $(n-t-1)(n-1)$.

4. Rotating Coordinator Paradigm

Another way to reduce message complexity is to use the *rotating coordinator paradigm* [5, 2], where in each round only the coordinator sends messages to other processes. Figure 3 presents a t-resilient consensus protocol for synchronous distributed systems. Consensus() is made up of $t+1$ rounds. Each round r ($1 \le r \le t+1$) is managed by a predetermined coordinator, p_r. In each round r, p_r will send v_r to every process whose ID is bigger than r. When a process p_j receives a value in this round, it will set v_j to the value.

The protocol described in Figure 3 solves the Uniform Consensus problem in synchronous systems where up to t processes can crash. It is easy to prove the correctness. Because at most t processes can crash, there is at least one round of the $t+1$ rounds in which the coordinators in those rounds do not crash. Assume r is the first round that the coordinator p_r does not crash in its round. According to the protocol, every process whose ID is bigger than r will set its value to v_r at the end of round r. And every process whose ID is smaller than r have crashed. Then after round r, all non-crashed processes maintain the same value.

```
Function Consensus(v_i)
r=0
while r < t + 1 do
   r = r + 1;
   if (i = r) then foreach j > i: send (v_i) to p_j;
   else let v be the value received from p_r during the rth round;
         v_i = v;
   endif;
end while
return (v_i)
```

Figure 3. the rotating coordinator paradigm

Cost. The time complexity is trivially $t+1$ rounds. During each round r, the round coordinator sends $n-r$ messages if not crash. Hence, the message complexity of the protocol is bounded by $\sum_{i=1}^{t+1}(n-i) = (t+1)(n-t/2-1)$. In existing non-rotating coordinator-based consensus protocols, every process needs broadcast messages in the first round, then the message cost in the first round is at least $(n-1) \times n$, which is more than $\sum_{i=1}^{t+1}(n-i)$. Thus, comparing to those protocols, the rotating coordinator protocol uses the minimum number of messages.

One shortcoming of the protocol is that it is not an early-stopping one. Another shortcoming is due to the rotating coordinator based protocols: this protocol just allows the coordinator to impose a value as the decided value in each round. Thus, it cannot solve GDC problem.

5. Extension to Solve the GDC Problem

In a distributed computation, a *Global Data* 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 of it, defines the Global Data Computing problem.

The distributed system consists of n processes, $\Pi = \{p_0, \ldots, p_{n-1}\}$. 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. Let GD_i denotes the local variable of p_i intended to contain the local copy of GD. The problem is formally specified by a set of four properties as following [8]. Let $\perp$ be a default value that will be used instead of the value v_j when the corresponding process p_j crashes prematurely.

- 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 ($GD_i[i]=v_i$).

Now explain how to change the proposed protocol in Section 3 to solve the GDC problem:

- In Figure 2a, change the pseudocode

  "**let** v = first non-$\perp$ value of V_i; **return**(v)"

  $\Downarrow$

  "**return**(V_i)"
- In Figure 2b, change pseudocodes

  "**let** v = first non-$\perp$ value of $V_{i,j}$ where $CP_j \in R_i(r)$; **return**(v)"

  $\Downarrow$

  "**return**($V_{i,j}$), where $CP_j \in R_i(r)$"

  "**let** v = first non-$\perp$ value of V_i; **return**(v)"

  $\Downarrow$

  "**return**(V_i)"

It is easy to prove the correctness. We just need to show the Obligation Property is achieved. When a *CP* decides, its proposed value is in the vector initially.

If a non-*CP* decides, there are two cases. First, the process decides based on its own vector, its value is in its vector initially. The second case, the process decides on a vector copy of a *CP*, because the non-*CP* does not crash, all non-crashed *CP*s in the first round must receive its value, the Obligation Property is achieved in this case also.

6. Discussion and Conclusion

In this paper we extend an early-stopping uniform consensus protocol for synchronous systems with crash failures proposed in [13] to reduce message complexity but still achieving the min(t + 1, f + 2)-rounds lower bound. Moreover, it can solve other related problems such as the GDC problem with simple revision.

The contribution of this paper is that the proposed protocol may be used in other uniform consensus protocols both for synchronous distributed systems or asynchronous systems. The extending condition is that before a process decides in a round, it must learn that it can do so in the previous round and broadcast the deciding flag in the current round. If the condition satisfied, the extended protocol just treat with t + 1 processes, the lower bound min(t + 1, f + 2) is still achieved. The rest processes can execute a procedure like Consensus2() in section 3, they just receive messages in each round and make decision when all *CP*s can make decision. The benefit of this method is that the message complexity is reduced while lower bound is still achieved.

Acknowledgement

This work is partially supported by the University Grant Council of Hong Kong under the CERG Grant B-Q518, the Hong Kong Polytechnic University under HK PolyU ICRG grant A-P202, and the National University of Singapore, under Academic Research Fund R-252-000-180-112.

References

[1] B. Charron-Bost, and A. Schiper, "Uniform consensus harder than consensus", Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[2] T. Chandra and S. Toueg, "Time and message efficient reliable broadcasts", Distributed Algorithms, 4th International Workshop, WDAG′90, Bari, Italy, Sep. 1990, Proceedings. Lecture Notes in Computer Science 486 Springer 1991, 289-303.

[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. Helary, and M. Raynal, "Early-stopping in Global Data Computation", Technical paper, IRISA, France, 2002.

[5] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the presence of partial synchrony". Journal of the ACM, vol. 35, no. 2, 1988, 288-323.

[6] D. Dolev, R. Reischuk, and R. Strong, "Early-stopping in Byzantine Agreement", J. ACM, vol. 37, no. 4, Apr. 1990, 720-741.

[7] 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.

[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] I. Keidar and S. Rajsbaum, "A Simple Proof of the Uniform Consensus Synchronous Lower Bound", Information Processing Letters, 85(1), 2003, 47-52.

[10] N. Lynch, "Distributed Algorithms", Morgan Kaufmann, 1996.

[11] 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.

[12] A. Mostéfaoui, S. Rajsbaum and M. Raynal, "Using Condition to Expedite Consensus in Synchronous Distributed Systems", Distributed Computing, Proceedings 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003.

[13] M. Raynal, "Consensus in Synchronous systems: A Concise Guided Tour", Proceedings of PRDC-9: 2002 Pacific Rim International Symposium on Dependable Computing, Tsukuba, Japan, 2002.