樹狀網路上找尋中心點的有效率錯誤抑制自我穩定演算法

黃哲志
元智大學資訊工程系
中壢市遠東路 135 號
cstetz@cs.yzu.edu.tw

林基成
元智大學資訊工程系
中壢市遠東路 135 號
csjclin@cs.yzu.edu.tw

周政佑
元智大學資訊工程系
中壢市遠東路 135 號

摘要

本篇論文提出一個在樹狀網路找中心點(centers)的自我穩定演算法，具有以下兩點特性：當系統發生 single transient fault，(1)只有 faulty node 會去修正資料，(2)stabilization time 爲 $O(min(\Delta^2,n))$，Δ爲 maximal nod degree，n 爲 nodes 的個數。若一分散式系統在任何起始狀態(initial states)下，皆能不經外力的介入，而在有限步驟之內恢復到正確的狀態，且一直維持在正確的狀態，我們稱這個系統爲自我穩定系統(self-stabilizing system)。實際上，分散式系統一次只有一個 node 發生 transient faul 的機率遠大於多個 nodes 同時發生 transient fault。到目前爲止在樹狀網路找中心點的自我穩定演算法，當系統發生single transient fault 時，會有 O(n)個nodes 的資料被影響，而stabilization time 爲 $O(n^2)$。

關鍵字：自我穩定演算法，中心點，錯誤抑制

1 簡介

分散式系統，一般來說是由多個獨立的電腦經由網路連結所構成，在這些電腦上執行的眾多processe 係經由網路彼此溝通來完成某一些特定的工作。當電腦裡的資料發生錯誤，系統可能就沒辦法正常運作，而由於分散式的特性，這些錯誤很難去察覺與修正。因此，自我穩定系統的研究在近十幾年蓬勃的發展。若一個系統在任何的起始狀態(initial states)下，皆能不經外力，而在有限步驟之內會到達正確的狀態，並且會一直維持在正確的狀態，我們稱這個系統具有自我穩定的性質。因此，在一個自我穩定系統，如果資料發生錯誤使得系統變成不正確的狀態，則在錯誤不繼續發生下，皆能不經外力，而在有限步驟之內會到達正確的狀態，並且會一直維持在正確的狀態。

自我穩定演算法雖可確保一旦系統遭受到 transient faults(即資料發生錯誤)，有限步驟的系統運作之後仍可恢復到正確狀態，可是它最大的缺點即是恢復時間往往過長，即使只有一個 process 發生 transient faults，系統仍然可能會經過很長的時間才恢復到正確狀態。所以系統就會有相當的時間是處於不正確的狀態，爲了要處理這種情況，[3]中提出了錯誤抑制之自我穩定演算法(Fault-containing self-stabilizing algorithm)，相較於傳統的自我穩定演算法，在只有一個 process 發生 transient fault 的情況下，錯誤抑制之自我穩定演算法能在相當快的時間內能夠恢復到正確狀態。

1.1 自我穩定演算法

我們可用一個連通且無方向性的圖$G = (V, E)$來表示分散式系統，V爲 G 中 nodes 所成的集合，E 爲 G 中 edges 所成的集合，每個 node 可視爲一個 process，而每個 edge 表示 processes 間的 links，nodes 間如有 edge 相連我們稱爲鄰居。一個分散式系統中，每一個 node 的狀態(local states)是由該 node 上所有變數的值所組合而成，而系統的狀態(global states)則是由所有的 nodes 的狀態組合而成。

一個分散式系統中，系統的狀態可分成合理狀態(legitimate states) 和不合理狀態 (illegitimate states)。若一個系統在任何的起始狀態下，皆能不經外力，系統保證在有限步驟之內恢復到合理狀態，並且在錯誤不繼續發生下，會一直維持在合理狀態，我們稱這個系統具有自我穩定的性質。而在自我穩定系統上執行的分散式演算法我們稱之爲自我穩定演算法。

自我穩定系統中，每個 node 上都有一個 node algorith ，而一個 node algorit h 是由一些 rules 所組成，每一個 rule 都是 *<guard> → <action>* 的形式，"*guard*"是由 node 本身的變數和其鄰居的變數所構成的 Boolea expression；

而 "*action*" 則是所要做的動作。一個 node 的某一個 guard 爲 true 我們稱這個 node 有 *privilege*。而當一個的 node 執行了一個 "*action*"，我們稱這個 node 做了一個 move。從不合理狀態到合理狀態所需之最長的時間，我們稱之爲 stabilization time。

但一個 node 如何知道鄰居的狀態呢？自我穩定系統有兩個模型分別爲 message passin 和 shared variable。在 message passin 的模型下，每個 node 可以將自己的狀態放在 message 內送給鄰居；而 shared variabl 的模型下，node 可以直接讀取鄰居的資料。本論文將採用 shared variable 的模型。

不同的 nodes 可能同時具有 privilege，在大多數自我穩定演算法的論文中都有一個假設機制稱爲 central demon，可以在所有具有 privilege 的 nodes 中隨意選取一個 node 去做 move，當這個 node 做完 move 之後，central demon 才可以再選取一個 node 去做 move，因此，central demon 可以使得有 privilege 的 nodes 以任意順序去循序執行。本論文也是採用 central demon 的假設機制。

自我穩定系統主要有以下兩個好處：

(1) 自我穩定系統不需要初始化，系統從任何一個狀態開始執行，都會到達合理狀態。
(2) 若系統發生了 transient fault 使系統的狀態變成不合理狀態，系統會自動的恢復到合理狀態。

自我穩定演算法分爲兩類，一、系統在合理狀態時，沒有 node 有 privilege；二、系統在合理狀態時，至少有一個 node 有 privilege。在本論文中所提出的找尋中心點演算法是屬於第一類。

1.2 錯誤抑制

大部分的自我穩定演算法中，系統狀態在合理狀態時，一個 node 發生錯誤(這種 fault 稱爲 single transient fault)，可能會造成鄰居有 privilege，如果 demon 選擇此鄰居做 move，則鄰居原本正確的資料就會被改錯，可能又會造成此鄰居的鄰居有 privilege，如此，錯誤就會擴散出去，導致更多的錯誤發生，系統可能就會經過很長的時間才恢復到合理狀態。所以系統就會有相當的時間是處於不合理狀態，爲了要處理這種情況 [3]中提出了錯誤抑制之自我穩定演算法 (Fault-containing self-stabilizing algorithm)，相較於傳統的自我穩定演算法，在只有一個 node 發生錯誤的情況下，錯誤抑制之自我穩定演算法能在相當快的時間內能夠恢復到合理狀態。

系統在合理狀態時，single transient fault 發生後使得系統變爲不合理狀態，我們定義這種狀態爲 1-faulty state。接下來我們對第一類的自我穩定演算法定義錯誤抑制。每一個 node 上的變數分成主要變數與次要變數兩種。

定義 1 [錯誤抑制之自我穩定演算法]
若一個自我穩定演算法從任何一個 1-faulty state 到達合理狀態都只有一個 node 會改變主要變數值，則此演算法稱爲錯誤抑制之自我穩定演算法。

本論文設計了一個可以在樹狀網路找中心點的自我穩定演算法，此演算法具有錯誤抑制的特性。

1.3 相關研究文獻

1.3.1 已發表過的相關文獻

1997 年 Gupta [3]在他的博士論文中提出三個具有有錯誤抑制特性之自我穩定演算法，分別爲 Leader election on oriented rings、Construction a spanning tre 、Construction a breadth-first-search tree，這三個演算法都有在學術會議中發表[4,5]。

1994 年 Karaata 等人提出了一個在樹狀網路尋找中心點的自我穩定演算法[6]，但是並沒有錯誤抑制的特性。此演算法從 1-faulty stat 到達合理狀態的過程中，最多會有 $O(n)$個 nodes 的資料會被修改並且在single fault 的情形下，stabilization time 爲 $O(n^2)$。我們所提的演算法是以[6]中的演算法爲基礎，因此我們將先介紹[6]中的演算法。

1.3.2 Karaata 等人的演算法

在[6]的演算法中，每個 node 上都只有一個變數 h，此變數可以被鄰居所讀取，但只能被自己更改。以下我們先介紹一些符號：

- $d(i, j)$ 表示 node i 和 node j 的距離，即 node i 和 node j 間最短路徑的長度。
- $e(i) = \max\{d(i, j) \mid i, j \in V\}$ 表示 node i 的離心率

(*eccentricity*)，即 node i 和離 i 最遠的 node 之距離。

- *center*(G)={ $i \in V \mid e(i) \le e(j)$ for all $j \in V$ }表示 G 的中心點所成的集合，即 G 中離心率最小的 nodes 所成的集合。

圖 1 中，node 4 的離心率最小，所以此樹的中心點爲 node 4。圖 2 中，node 4 和 node 5 的離心率最小，所以此樹的中心點爲 node 4 和 node 5。

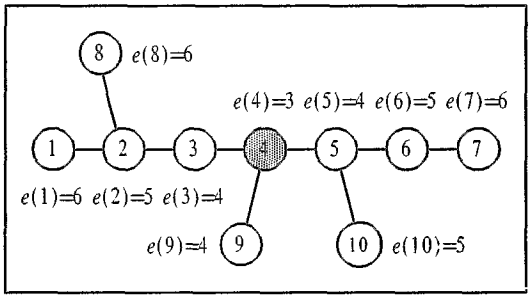

圖 1. 具有單一中心點的樹狀圖

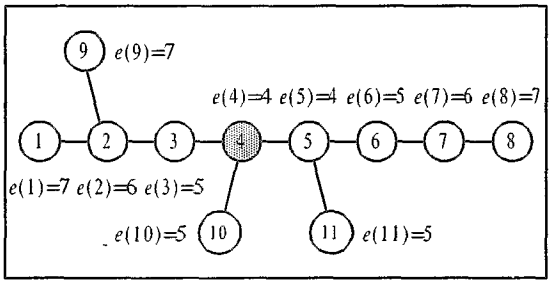

圖 2. 具有雙中心點的樹狀圖

我們用 $h.i$ 來表示 node i 中變數 h 的值，接下來，我們再介紹幾個符號：

- $N.i = \{ j \in V \mid (i, j) \in E \}$ 表示 node i 所有鄰居所成的集合。
- $N_h.i = \{h.j \mid j \in N.i\}$ 表示 node i 所有鄰居的 h 值所成的 multi-set。
- $N_h^-.i = N_h.i - \{\max N_h.i\}$ 表示 $N_h.i$ 中扣掉一個最大 h 值所成的 multi-set。例如：若 $N_h.i$={5, 5, 4}，則 $N_h^-.i$ = {5, 4}。

我們接下來介紹與 $h.i$ 有關的一個條件，我們稱之爲 *high-condition*：

$$h.i = \begin{cases} 0 & \text{if } i \text{ is a leaf} \\ 1 + \max N_h^-.i & \text{otherwise} \end{cases} \quad (1)$$

其中 $\max N_h^-.i$ 表示 $N_h^-.i$ 中的最大值。

此系統的合理狀態爲所有的 nodes 都滿足 *high-condition*；反之，則爲不合理狀態。在合理狀態下中心點爲 h 值大於或等於所有鄰居 h 值的 nodes。

圖 3 中所有 nodes 都滿足 *high-condition*。中心點爲 node 4 和 node 5。

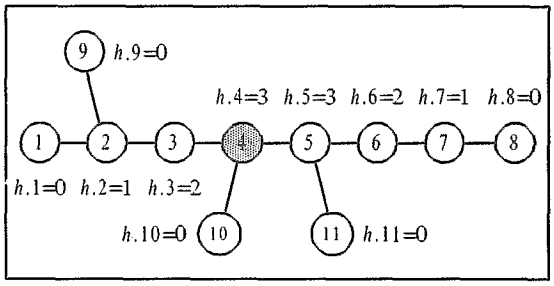

圖 3. 所有 node 都滿足 *high-conditi o* 的樹狀圖

Karaat 的演算法：

Ra: (i is a leaf) $\wedge\ h.i \neq 0 \quad \rightarrow \quad h.i := 0$

Rb: (i is not a leaf) $\wedge\ h.i \neq 1 + \max N_h^-.i \quad \rightarrow \quad h.i := 1 + \max N_h^-.i$

此演算法不具備錯誤抑制的性質。考慮一個樹狀圖 G 如圖 4 所示。此時系統是在合理狀態，若只有 node 1 發生錯誤使得 $h.1$=8，此時的系統狀態如 5 所示，node 1 滿足 *Ra* 的 guard，而 node 2 滿足 *Rb* 的 guard。

圖 4

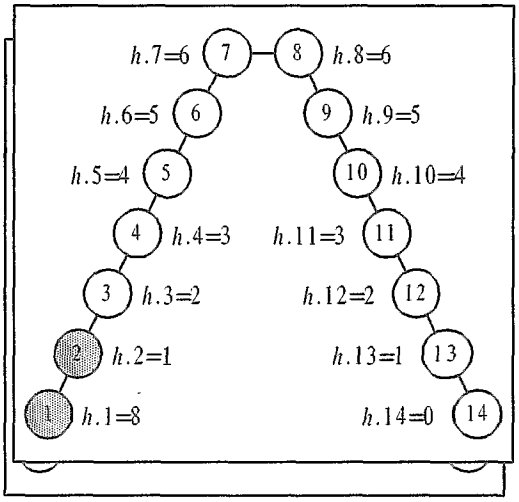

圖 5

Demon 選擇 node 2 move

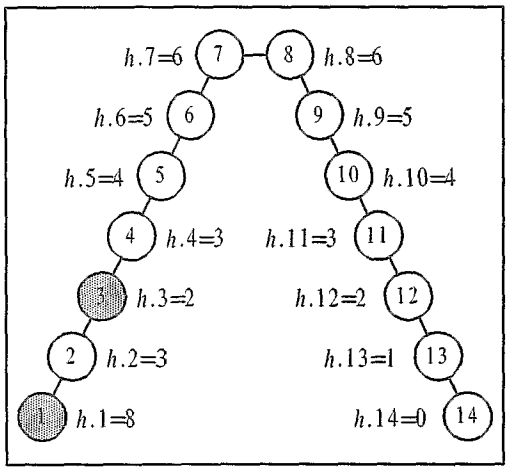

圖 6

Demon 選擇 node 3 move

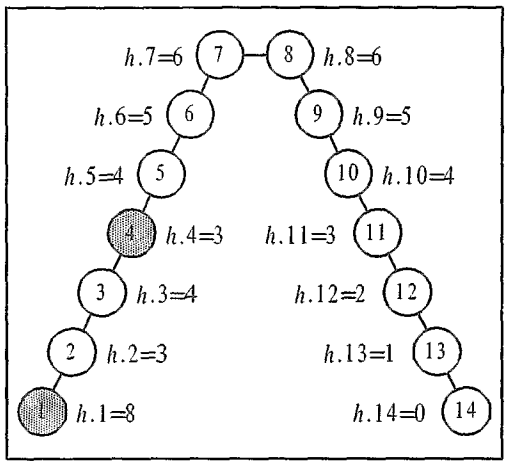

圖 7

Demon 選擇 node 4 move

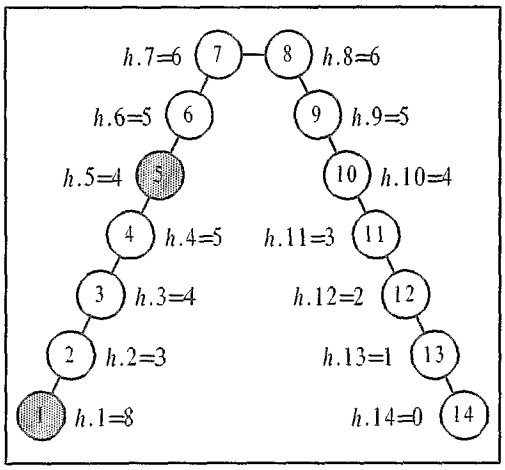

圖 8

Demon 選擇 node 5 move

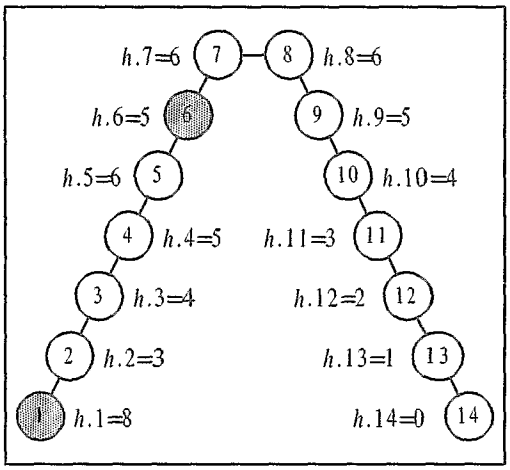

圖 9

Demon 選擇 node 6 move

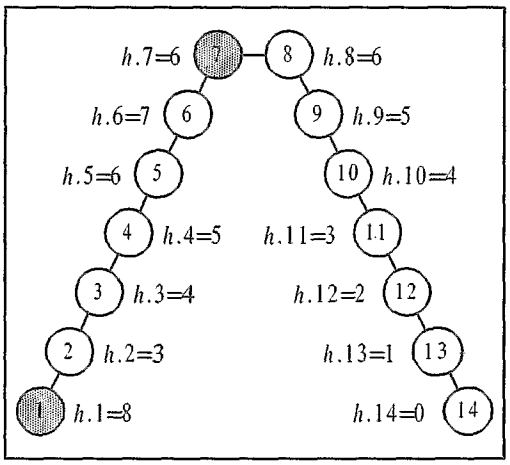

圖 10

Demon 選擇 node 7 move

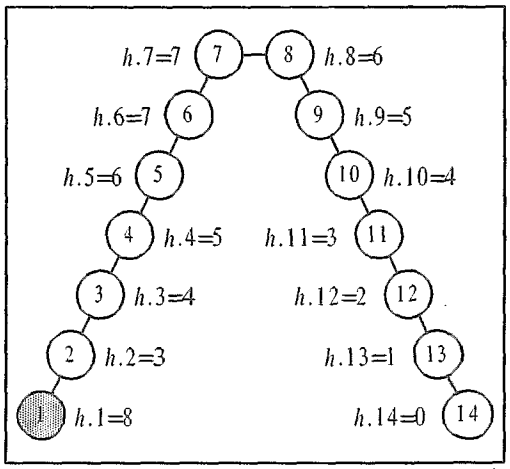

圖 11

從圖 4 到圖 11 我們知道 single transient fault 發生後可能會造成 O(n)個 nodes 的 h 值改變，所以 Karaata 的演算法不具有錯誤抑制之特性，本論文所提出的演算法將會使得節點 2、3、4、5、6、7 不會去修改 h 值，而只有節點 1 會執行 *Rb* 將 $h.1$ 設爲 0。

2 理論基礎

若系統的狀態是 1-faulty state(即系統在合理狀態時，只有一個 node 發生錯誤)，可能會造成 faulty nod 的鄰居有 privilege，如果 demon 選擇此鄰居 move，則此鄰居原本正確的資料就會被改錯，可能又會造成此鄰居的鄰居有 privilege，如此，錯誤就會擴散出去，導致更多的錯誤發生。本論文主要的idea 就是要設法讓 node 可以"察覺"是否自己發生錯誤，若是自己發生錯誤，才進行修正。我們定義三個 Boolean functions 如下：

$Ga.i \equiv (i \text{ is a leaf}) \wedge h.i \neq 0$

$Gb.i \equiv (i \text{ is not a leaf}) \wedge h.i \neq 1 + \max N_h^{-}.i$

$G.i \equiv Ga.i \vee Gb.i$

當系統的狀態爲 1-faulty stat 時，我們對任何一個 node i 考慮以下三個 cases：

1) $G.i$ 爲 true 且 node i 有兩個鄰居 j、k 使 $G.j$ 及 $G.k$ 爲 true；
2) $G.i$ 爲 true 且 node i 恰有一個鄰居 j 使 $G.j$ 爲 true；
3) $G.i$ 爲 true 且對 node i 所有的鄰居 j，$G.j$ 都爲 false。

先考慮 Case 1)，我們可得定理 1。

定理 1：設系統的狀態是 1-faulty state，若存在一個 node i 使 $G.i$ 爲 true 且 node i 有兩個鄰居 j、k 使 $G.j$ 及 $G.k$ 爲 true，則錯誤發生在 node i。

證明：參見圖 12，設系統的狀態是 1-faulty state，若存在一個 node i 的 $G.i$ 爲 true 且 node i 有兩個以上的鄰居 $i_1, i_2, \ldots, i_k$ 使 $G.i_1, G.i_2, \ldots, G.i_k$ 爲 true，$k \geq 2$，則發生錯誤的 node 一定在這些 nodes 當中。若錯誤發生在 node i_1，則不會使得 nodes $i_2, \ldots, i_k$ 的 $G.i_2, \ldots, G.i_k$ 爲 true，因爲 G 值爲 true 或 false 只和自己的鄰居有關，與假設矛盾，所以錯誤不會發生在node i_1 上；同理，錯誤也不會發生在 nodes $i_2, \ldots, i_k$。所以錯誤發生在 node i。

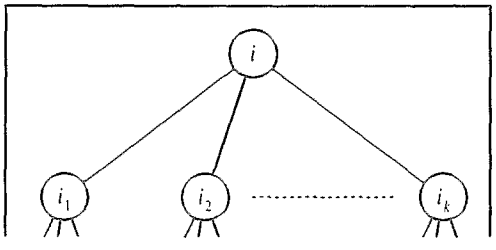

圖 12. 系統的狀態是 1-faulty state， node i 的 $G.i$ 爲 true 且 node i 有兩個鄰居 $i_1, i_2, \ldots, i_k$ 使 $G.i_1, G.i_2, \ldots, G.i_k$ 爲 true。

再考慮 Case 2)，我們可得定理 2。

定理 2：設系統的狀態是 1-faulty stat 且 node i 和 node j 互爲鄰居且 $G.i$ 和 $G.j$ 都爲 true，若 $[(h.j \geq h.i) \vee (\forall k \in N.i - \{j\} : h.j \geq h.k)] \wedge [(h.j < h.i) \vee (\exists k \in N.j - \{i\} : h.k \geq h.i)]$，則錯誤發生在 node i。

證明：省略。

Case 3) 當系統的狀態是 1-faulty stat 時，若 node i 的 $G.i$ 爲 true 且 node i 所有鄰居 j 的 $G.j$ 都爲 false，則顯然錯誤發生在 node i。

3 我們的演算法

因爲 node i 無法知道其鄰居 node j 的 $G.j$ 是 true 還是 false，所以我們使用了問答的機制，所謂的問答機制即爲每個 node 上增加兩個變數分別爲 q (question)和 a (answer)。q 有兩個值分別爲 0 和 1；a 有三個值分別爲⊥，0 和 1。我們分別用 $q.i$ 和 $a.i$ 來表示 node i 中的變數 q 和 a。當 $q.i = 1$ 時，表示 node i 去問它的鄰居 j：$G.j$ 爲 true 還是 false；$q.i = 0$ 時，表示 node i 沒有問。當 $a.i = \perp$ 時，表示 node i 沒有回答任何問題；當 $a.i = 1$ 時，表示 node i 回答 $G.i$ 爲 true；當 $a.i = 0$ 時，表示 node i 回答 $G.i$ 爲 false。

對於 $[(h.j \geq h.i) \vee (\forall k \in N.i - \{j\}, h.j \geq h.k)] \wedge [(h.j < h.i) \vee (\exists k \in N.j - \{i\}, h.k \geq h.i)]$，node i 無法知道 node j 的鄰居的 h 值，故無法判斷 $\exists k \in N.j - \{i\}, h.k \geq h.i$ 爲 true 還是 false，所以我們就在每個 node 中再增加一個變數 c，提供資料給其它的 nodes，我們用 $c.i$ 來表示 node i 中變數 c 的值。

定義 2: [合理狀態] 本系統的合理狀態爲所 node i 都滿足 *high-condition* (1)且 $q.i = 0$，$a.i = \perp$，$c.i = 0$。

我們的演算法如下：

R1: $Ga.i \rightarrow h.i := 0$

R2: $Gb.i \wedge (q.i = 0) \wedge [(\forall j \in N.i : a.j = \perp) \vee (\exists j,k \in N.i : j \neq k : q.j = 1 \wedge$

$a.k \neq \perp)] \rightarrow q.i := 1$

R3: $Gb.i \wedge (q.i=1) \wedge (\forall j \in N.i : a.j=0) \rightarrow h.i := 1 + \max N_h^{-}.i$

R4: $Gb.i \wedge (q.i=1) \wedge (\left|\{ j \in N.i \mid a.j = 1\}\right| \geq 2)$

$\rightarrow h.i := 1 + \max N_h^{-}.i$

R5: $Gb.i \wedge (q.i=1) \wedge (\exists! j \in N.i : a.j=1) \wedge$

$\{\{[(h.j \geq h.i) \vee (\forall k \in N.i - \{j\} : h.j \geq h.k)] \wedge [(h.j < h.i) \vee (c.j=1)]\} \vee$

$[(h.i=h.j) \wedge (q.j=1) \wedge (a.i=1) \wedge (c.i=0) \wedge (c.j=0)]\}$

$\rightarrow h.i := 1 + \max N_h^{-}.i$

R6: $(a.i \neq f.i) \vee (c.i \neq g.i) \rightarrow a.i := f.i;\ c.i := g.i$

R7: $[\neg Gb.i \vee (i \text{ is a leaf})] \wedge (q.i=1) \rightarrow q.i := 0$

其中

$$f.i = \begin{cases} \perp & \text{if } (\forall j \in N.i : q.j = 0) \\ 1 & \text{if } (\exists j \in N.i : q.j = 1) \wedge G.i \\ 0 & \text{if } (\exists j \in N.i : q.j = 1) \wedge \neg G.i \end{cases}$$

$$g.i = \begin{cases} 1 & \text{if } (\exists! j \in N.i : q.j = 1) \wedge (\exists k \in N.i - \{j\} : h.k \geq h.j) \\ 0 & \text{otherwise} \end{cases}$$

4 正確性

演算法的正確性分爲兩個部分討論，一、證明我們的演算法爲自我穩定演算法；二、證明我們的演算法具有錯誤抑制之特性。因爲以下定理的證明都很長，所以我們在此都將予以省略。

4.1 自我穩定

定理 3：當系統在不合理狀態時，至少存在一個 node 有 privilege。

定理 4：系統從任何一個起始狀態開始執行，有限步驟之內一定會停。

定理 5：本演算法爲自我穩定演算法。

4.2 錯誤抑制

定理 6：系統在 single fault 發生後，faulty node 以外的任何 node 皆不會改變主要變數 h 的值。

因此我們的演算法符合定義 1，即從任何一個 1-fault state 到合理狀態之間，只有一個 node 會改變主要變數值。

定理 7：在 single fault 情況下，本演算法的 stabilization tim 爲 $O(\min(\Delta^2, n))$。

5 結論

我們提出了一個樹狀網路上找尋中心點的錯誤抑制自我穩定演算法，不管一開始系統的初始值如何以及發生任何錯誤讓資料出錯，都會在執行一段時間過後到達合理狀態；若是發生單一節點錯誤時，錯誤不會擴散到其他節點上且在 $O(\min(\Delta^2, n))$ 步之內就會到達合理狀態。

參考文獻：

[1] E. W. Dijkstra, "Self-Stabilizing System in Spite of Distributed Control," Communication of ACM, Vol. 17, No. 11, pp. 643–644, 1974.

[2] E. W. Dijkstra, "A Belated Proof of Self-Stabilization," Distributed Computing, Vol. 1, No. 1, pp. 5–6, 1986.

[3] Arobinda Gupta, "Fault-Containment in Self-stabilizing Distributed Systems," Ph.d. thesis, University of Iowa, Santa Cruz, 1997.

[4] Sukumar Ghosh, Arobinda Gupta and Sri ram V. Pemmaraju, "A Fault-Containing Self-Stabilizin Algorithm for Spanning Trees," Journal of Computin and Information, Vol. 2, No. 1, pp. 322 –338, 1996. (Special issue on the English International Conference of Computing and Information, Waterloo, Ont ario, Canada, June 1996.)

[5] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju, "An Exercise in Fault-Containment: Leader Election on Rings," Information Processin Letters,Vol. 59, pp. 281 –288, 1996.

[6] M. H. Karaata, S. V. Pemmaraju, S. C. Bruueil, S.Ghosh, "Self-Stabilizing Algorithms for Findin Centers and Medians of Trees," In PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 374, 1994.

[7] M. H. Karaata, S. V. Pemmaraju, S. C. Bruueil, S Ghosh, "Self-Stabilizing Algorithms for Findin Centers and Medians of Trees," Technical Report, TR94–03, University of Iowa, 1994.

[8] M. Schneider, "Self-stabilization," Computin

Surveys, Vol. 25, No. 1, pp. 45–67, 1993.

[9] L. C. Wu, S. T. Huang, "Distributed Self-Stabilizin Systems," Journal of Information Science and Engineering,Vol. 11, pp. 307 –319, 1995.