國 立 交 通 大 學

電信工程研究所

碩 士 論 文

利用方案驅動方式討論乙太網路連結錯誤管理協定的開發

Scenario-Driven development for Ethernet Connectivity Fault Management Protocol

研 究 生：廖振翔　　Student: Chen-Hsiang Liao

指導教授：田伯隆 博士　　Advisor: Dr. Po-Lung Tien

中華民國九十九年十月十八日

利用方案驅動方式討論乙太網路連結錯誤管理協定的開發

Scenario-Driven development for Ethernet Connectivity Fault Management Protocol

研 究 生：廖振翔　　Student: Chen-Hsiang Liao

指導教授：田伯隆 博士　　Advisor: Dr. Po-Lung Tien

國 立 交 通 大 學
電信工程研究所
碩 士 論 文

A Thesis Submitted to
the Department of Communication Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
In
Communication Engineering
Oct. 2010
Hsinchu, Taiwan, Republic of China

中華民國九十九年十月十八日

利用方案驅動方式討論乙太網路連結錯誤管理協定的開發

研 究 生：廖振翔　　指導教授：田伯隆 博士

國 立 交 通 大 學

電信工程研究所

中文摘要

UML 是一種程式開發的觀念。對於開發大型程式可以有效的減低錯誤發生的機率，以及提升整體程式開發的效率。我們使用的 Scenario-Driven 的流程，是一種程式開發的流程，也就是先了解程式的運作方式，寫出使用實例，建立物件彼此之間的關係，做出時序圖，將時序圖轉換成各個物件的狀態圖，最後完成程式開發的一個程序。

我們針對 802.1ag 標準來做一些探討的動作，802.1ag 是 IEEE 用於 Carrier Ethernet 上的一個連結性的錯誤管理協定，提供了路徑發現、錯誤檢測、錯誤確認和隔離、錯誤通知以及錯誤修復。

我們主要探討如何將 802.1ag 標準內表示的狀態圖轉換成我們所需要的程式碼、對於訊號事件的處理以及利用排程器處理時間事件的方式，最後針對 Continuity Check Message(CCM)，做狀態機處理封包的時間量測，用以驗證使用 Scenario-Driven 方法在電腦上的效能及穩定度。

Scenario-Driven development for Ethernet Connectivity Fault Management Protocol

Student: Chen-Hsiang Liao Advisor: Dr. Po-Lung Tien

Department of Communication Engineering
National Chiao Tung University

Abstract

UML is a concept of software engineering. It can decrease the probability of fault occurring and increase the efficiency of program development. We use Scenario-Driven, which is a procedure of program engineering, namely, we have to realize how the program working, write down the program use case, construct the relation between objects, build the sequence diagram, and then transfer the sequence diagram into object state charts. Finally, we finished the program.

In this thesis, our discussion aims at 802.1ag. 802.1ag is a protocol, which is used in Carrier Ethernet for connectivity fault management. It offers Path Discovery, Fault Detection, Fault Verification and isolation, Fault Notification, and Fault Recovery.

For the most part, we discussed how to transfer the state charts in 802.1ag into our code, how to deal with Signal Event and use Real-Time Scheduler to handle Time Event. The rest of this thesis, we measured the time that state machine deals with Continuity Check Message (CCM). Finally, the statistics can justify the efficiency and stability of Scenario-Driven procedure on computer.

誌謝

首先要感謝我的指導教授，廖維國老師。老師淵博的學問以及耐心、細心的指導，讓我在研究路上獲益良多。其次感謝田伯隆老師以及范國寶博士，在百忙之中抽空擔任我的口試委員，同時也給我諸多寶貴的建議。

首先，我要感謝我的姊姊，一路走來，姊姊對我的教誨甚嚴，幫助甚多。課業上，面對問題應該有的態度以及處理，使我對於問題能夠勇於面對，我銘感五內。

我還要感謝 812B 實驗室的學長們，Moon、Kemp、Wia 以及梓洋學長，給予我很多學業上的幫助與鼓勵。另外還有已經畢業的超哥、大嫂、賢宗、俊宏學長、Oga 學長以及郁媛學姐，你們對我的鼓勵與關心我都銘記在心。首特公司的林文隆學長，對於我研究上以及程式能力的提昇我也至上最高的謝意。蔡昂勳學長，你對我的鼓勵以及課業上的幫助我也必須致謝。還有學弟妹們，你們讓實驗室充滿了許多珍貴的回憶。還有兼賢，你給予我的諸多的建議我也非常感謝。

對於我的大學同學以及研究所好友們，還有以前成功高中幫助我的同學們，真的很謝謝你們。

我的女朋友麗淇，在我研究所路程也給予我很大的力量與鼓勵，在此也要致上謝意。

最後我要感謝我的爸爸與媽媽，您們在背後對我的支持是我在學校最大的力量，再次感謝您們。

目錄

圖目錄

第一章

序論

UML，Unified Modeling Language，是利用圖形化來達成快速程式開發的一種思維。利用 UML 的觀念以及我們所使用的 UML 工具輔助，可以迅速的將圖形轉變成程式開發人員所需要的程式碼。

在這篇論文內，我們使用 Scenario-Driven 的方式，可以詳細的描述出程式的特性並且迅速的完成所需開發的程式，最後更可以利用 Scenario，也就式程式運行所產生的時序圖，進一步的驗證程式的正確性，完成 Scenario-Driven 這套方法。

首先，程式開發人員必須要了解程式的運作方式，並且依據運作方式寫出 Use Case。再者，從 Use Case 當中，可以將大型程式所需要的物件(Object)規劃出來，並且連結各個物件彼此之間的關係(Relation)，也就是將指標指到對應的物件，此時我們需要用到物件模組圖(Object Modeling Diagram)來達成這件事情。再來，將時序圖(Sequence Diagram)做出，時序圖代表的就是物件在時間軸上的行為。接著，從完善的時序圖之中，便可以推出程式所需要的狀態圖(State Chart)，其所代表的意義是一個物件整體的行為，一個物件僅有一個狀態圖。只要推出程式所需要的狀態圖，程式開發人員就可以快速的將程式開發完成。

利用上述的程序，也就式 Scenario-Driven 的流程來開發軟體，可以避免因大型程式本身內部錯綜複雜的關係而發生不可預期的錯誤，UML 本身就是一個由大至小的規劃，也就是產生大型程式所有的框架後，再去處理細節的部份，對於整體程式的開發會非常的有條理及幫助。

在我們的實驗中，UML 也存在著些許問題，除了此種思維模式需要較長時間的訓練之外，我們使用的 UML 工具產生出來的程式碼會很龐大，針對第二個

問題，我們以圖 1.1 來做說明。

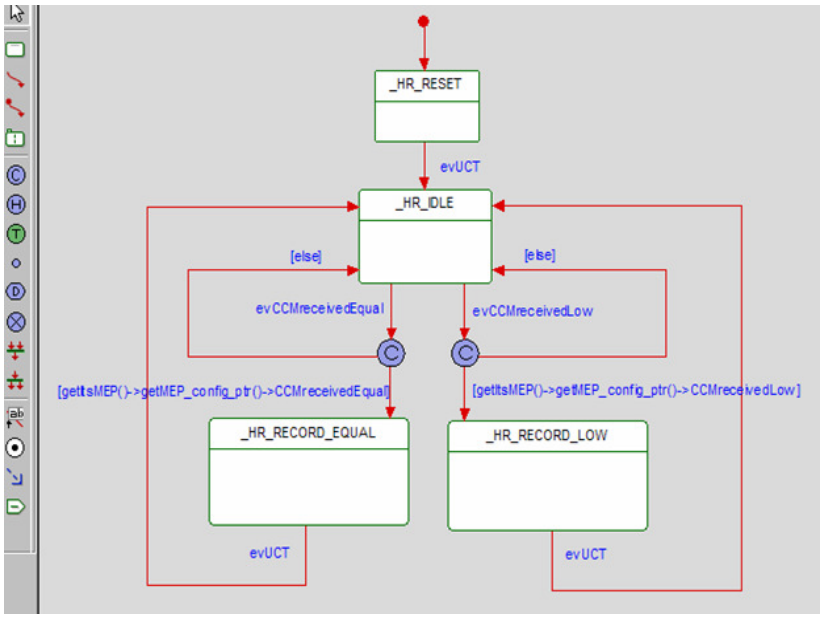

圖 1-1 狀態圖說明

我們利用圖 1-1 的狀態圖，比較我們所使用的工具產生出來的程式碼，與自己改良後程式碼的實際大小，發現差距甚大。原因在於使用的工具必須要引入相當多的表頭檔來處理狀態圖中的事件，進而符合各式各樣的大型程式。而我們在對於事件的處理方式則排除了引入大量的表頭檔。程式開發人員在使用工具同時，必須要注意到這方面的細節，將程式最佳化，去除不必要的程式碼進而完成自己所需要的最佳化程式。

這篇論文的主要目的，在於討論利用 UML 思維和所使用的 UML 工具輔助下，對於開發協定的益處，以及討論對於訊號事件(Signal Event)的處理和利用排程器(Scheduler)來處理時間事件(Time Event)的方式。我們將針對 802.1ag 這個標準做上述探討。

802.1ag 這套標準，主要是在偵測、定位並且隔離網路中錯誤發生的位置，主要由三個子協定所組成，分別為 Continuity Check、Loopback 以及 Linktrace 協定。

在我們的實驗中，完成了 Active Type Demultiplexer、Active Level Demultiplexer、Equal Opcode Demultiplexer、MEP Continuity Check 以及 MEP

Continuity Check Receiver 這幾個狀態機，並且測出了處理一個 Continuity Check 封包的時間，由圖 1-2 可以清楚說明我們完成的部份。

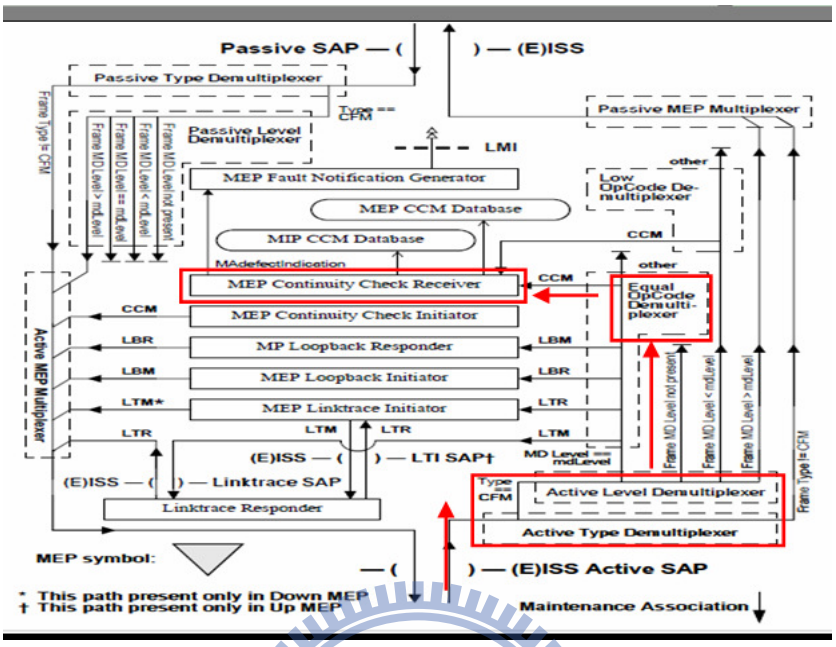

圖 1-2 實驗完成部份說明

由圖 1-2 是由 802.1ag 所定義的一個 Maintenance Association End Point(MEP)，MEP 所要做的事情便是接收、發送以及處理 CFM 型別的封包進而達成 802.1ag 三個子協定的功能。

一個封包從 Active Type Demultiplexer 接收進來，經過處理後進入到 Active Level Demultiplexer，再到 Equal Opcode Demultiplexer，判斷是哪一種型別的 CFM 封包，最後傳送到 MEP Continuity Check Receiver 做處理，我們測量的便是這個流程的時間。

論文的剩餘部份，在第二章，我們介紹 802.1ag 這個協定概要。在第三章，我們描述如何做物件規劃(Object Plan)、如何將 802.1ag 標準上面的狀態圖轉換成我們最後使用的程式碼以及對於訊號事件、時間事件的處理方式。第四章則是陳述我們量測的方法、量測處理 Continuity Check Message 的測試架構以及實驗環境下的參數設定。第五章則將實驗數據表示出來。最後，在第六章給予結論。

第二章

802.1ag 協定概要

2.1 802.1ag

802.1ag 是 IEEE 提出了一個用於 Carrier Ethernet 中進行錯誤管理的一個標準，即連結錯誤管理(CFM, Connectivity Fault Management)。由於在 Carrier Ethernet 之中，並沒有一個方法可提供錯誤偵測或錯誤確認的方法。不論在 VDSL 或是 ADSL 之中都有線路檢測的方法，好比 VDSL 之中的 SELT 以及 DELT 等等。唯獨存取網路(Access Network)部份缺少了偵錯的方式，所以 IEEE 定義了 802.1ag 來加強這一部份。802.1ag 提供了下列五項功能：

- 路徑發現(Path Discovery)
- 錯誤偵測(Fault Detection)
- 錯誤確認與隔離(Fault verification and isolation)
- 錯誤通知(Fault Notification)
- 錯誤修復(Fault Recovery)

其中的錯誤修復功能，必須要聯合其他協定一起來實行，故不在 802.1ag 的討論範圍之內。上述功能，主要由三個子協定來共同達成，分別為：

- Continuity Check Protocol
- Loopback Protocol
- Linktrace Protocol

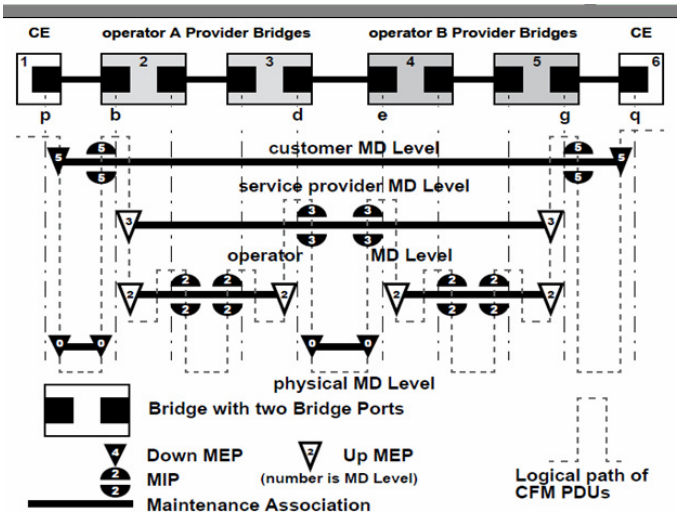

圖 2-1 802.1ag 概念

圖 2-1 我們簡單的說明 802.1ag 的概念，802.1ag 會將所要管理的網域利用 Maintenance Domain Level(MD Level)分層。每個層級裡面又會有許多的 Maintenance Association。

每個 Maintenance Domain 的網管人員只可以看到自己的 Domain，底層可以為它直接相鄰的上層提供服務，由圖中所示，customer MD Level 檢測出了問題之後，網管人員只知道本層的問題，他可以得知是 customer 到 service provider 之間出了問題，或是 service provider 本身出了問題。如果是後者，網路管理員無法進一步的定位具體的問題所在。所以更精確的定位就交給了 service provider。同樣的，service provider 橫跨了多個 operator domain，網路管理員只可以得知 service provider 和 operator 之間出了問題，要進一步定位出哪一個 operator domain 出了問題就要再往下交給 operator domain 來做偵測以及管理。802.1ag 就是使用這樣的概念，有效的將錯誤範圍縮小並精準定位出來。值得注意的是，圖 2-1 僅為一個簡單的範例，助於了解概念，對於 CFM 封包的走向無法藉由本圖完整了解。

接著，我們將說明 802.1ag 如何運作：

- 對網路的設備做最完善的規劃以及去設定 CFM 的元件。包含了設定不同使用者所涵蓋的 Maintenance Domain，再根據不同的 Maintenance Domain 設定

所涵蓋的 Maintenance Association，最後要根據不同的使用者設定對應交換器(Switch)上面的連接埠(Port)。

- 在第一個部份設定完善之後，便可以使用三個子協定來定位錯誤發生的地點。

由於所要管理的網域可能會很龐大，網路元件會很多，所以不論在 Maintenance Domain 的設置，Maintenance Association 的規劃以至於 port 與 MEP 之間的對應都是相當重要的。所以在整個 CFM 功能運作之前，就要做好最完善的規劃，之後網管人員維護時及協定運作才不會出現錯誤狀況。

2.2 Continuity Check Protocol

Continuity Check Protocol 由 MEP 裡面的 MEP Continuity Check Initiator 以及 MEP Continuity Check Receiver 一起共同完成。

針對 Continuity Check Protocol，也就是錯誤偵測的功能，我們可以將一個維護連結看成一群維護終點(MEP, Maintenance association End Point)的結合。裡面所有的 MEP 都擁有相同的 MAID(Maintenance Association Identitifer)，以及 MD Level，各個 MEP 就是藉由在該 MD Level 以及該 MAID 之中唯一的 MEPID(Maintenance association End Point Identitifer)來設定。所有在同一個 MA 的 MEPID 都會被一個完整的目錄所管理。

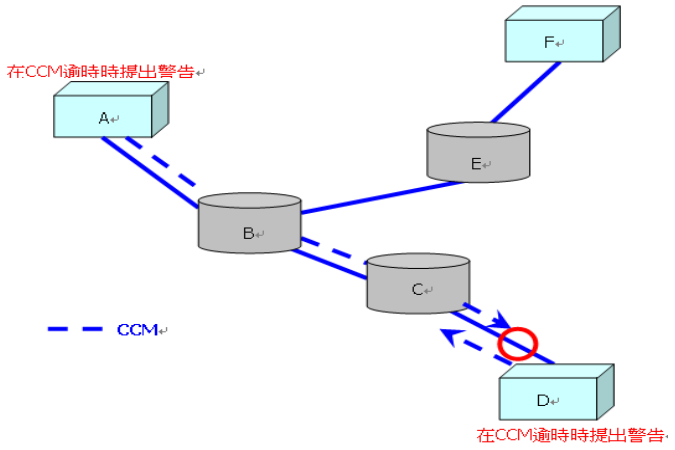

圖 2-2 Continuity Check Protocol

圖 2-2 可以清楚看到，假設由 MEP A、MIP B、MIP C 以及 MEP D 構成一個 MA，CCM 封包將各自由 MEP A 以及 MEP D 由 multi-cast 形式發送出去，並僅限於在 A，B，C，D 這個 MA 之中傳送。至於圖中 CCM 逾時的警告部分，在 802.1ag 裡面並沒有詳細說明，這部份應由程式設計人員自己規範合理時間。

2.2.1 Continuity Check Message

CCM(Continuity Check Message)提供可以偵測一個 MA 之內連結錯誤的方法，MA 當中的各個 MEP 可以被設定為週期性的傳送 CCM 封包。另外，CCM 為一個 multi-cast 的封包，destination_address 參數要根據這個 MEP 所存在的 MD Level 而定。在 802.1ag 標準內[1]規定它所應該負載的內容。

2.2.2 Connectivity Failures

連結錯誤在 802.1ag 裡面有所定義：

- MEP 本身或者網路連結錯誤：一個 MEP 沒有收到同一個 MA 之中任意其他 MEP 連續三個封包。
- 設置上的錯誤：MEP 收到一個 CCM 封包但是裡面的 transmission interval 不正確(同一個 MA 裡面，眾多 MEP 的 CCMinterval 是相同的)。
- 設置上的錯誤或是交叉連結(Cross Connection)錯誤：MEP 收到的 CCM 封包內有著不正確的 MAID 或者 MEPID，或者收到 CCM 的 MD Level 低於接收 MEP 的 MD Level。
- 交換機上的連接埠或者聚集連接埠(Aggregated Port)有錯誤：由 MEP 接收到的 CCM 封包裡面的 Port Status TLV 或者 Interface Status TLV 會陳述(Port Status TLV 或者 Interface Status TLV 為封包內的參數，在此不多做說明)。

2.2.3 RDI bit in Continuity Check Message

RDI(Remote Defect Indication)，會被 CCM 所負載。當一個 CCM 封包裡面沒有 RDI 表示傳輸的 MEP 正在從其他 MEP 接收 CCM 封包。

這個位元設計的原因，主要在於 CCM 是不會被回應的封包，這樣會造成一個很大的問題。一個 MA 之中，存在著許多 MEP，分別標示為 MEP 1，MEP 2，MEP 3…，如果 MEP 1 存在著單向連結(無法送達，可以接收)的問題，其他的 MEP 可以察覺到 MEP 1 的問題所在，但是 MEP 1 自己本身卻不知道問題已經發生。有鑒於此，RDI 這個位元就可以解決這樣的問題。MEP 1 在這種情況下，將會接收到其他所有 MEP 的 CCM 封包內都帶有 RDI 這個位元(數值為 1)。

如此一來，網路管理人員便可以檢驗屬於一個 MA 之中任何單一的 MEP，以及發現 MA 中任何一個 MEP 正在偵測到一個錯誤，以及哪些 MEP 有錯誤發生。

2.2.4 Sequence number in Continuity Check Message

每一個 CCM 封包裡面都要被傳輸，它的目的在於接收的 MEP 需要用這個 Sequence Number 來檢測以及計算遺失的 CCM 封包數目。

2.3 Loopback Protocol

Loopback Protocol 是由 MEP Loopback Initiator 以及 MP Loopback Responder 共同完成。其中 MP Loopback Responder 這個狀態機在 MEP 以及 MHF 裡面都存在著，而 MEP Loopback Initiator 只存在 MEP 之中。

2.3.1 Loopback Message，Loopback Reply

LBM(Loopback Message)是用來做 Fault Verification 以及 Fault Isolation 用的，它可以驗證眾多 MEP 以及 MIP 之間的連結性，一個 MEP 可以被網路管理

人員命令發送一個或多個 LBM 封包。另外，LBM 是屬於 unicast 的封包。

LBM 裡面存在著許多參數，皆需要在傳送前被 MEP 給初始化，其中 destincation_address 參數是跟傳送 LBM 的 MEP 在同樣 MA 裡面的另一個 MP 的 MAC Address。

另外，經由 MP Loopback Responder 收到 LBM 封包的 MP，都要驗證 LBM 封包是否有效(驗證方法在 802.1ag 的標準內有提及)，如果驗證 LBM 為有效的封包，那接收到 LBM 的 MP 就會回應一個 unicast 的 LBR(Loopback Reply)封包回到原來的 MEP，由圖 2-3 我們可以有更清楚的說明。

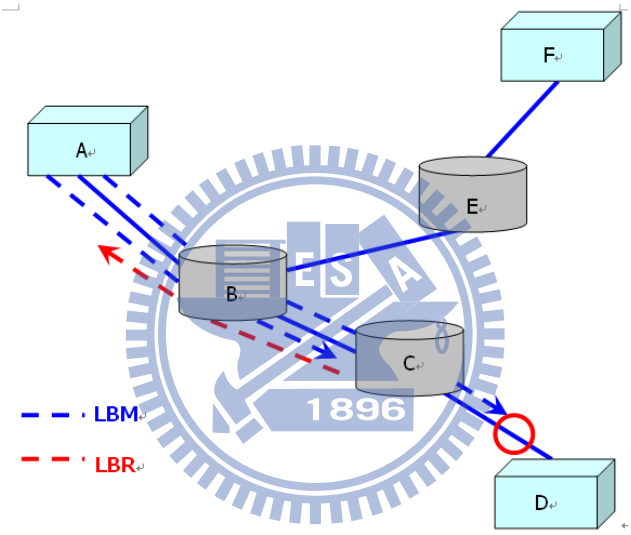

圖 2-3 Loopback Protocol

由圖 2-3 所示，Loopback 的作法其實就是針對某一個 MEP(Port)做偵測，網路管理人員可以直接的測試從 A 這個 MEP 到 D 這個 MEP 的網路狀況。我們由 MEP A 傳送 LBM 到 MIP C，則 A 可以收到 C 的 LBR 封包(由於 Loopback 和 Linktace 的方式不一樣，所以 MIP B 並不會回應 LBR)。接著，我們再由 A 傳送 LBM 到 MEP D，這時候 A 卻無法收到 D 回傳的 LBR，因為錯誤就可以定位出是 C 與 D 的連結上面出了問題。

值得一提的是，當 LBM 數量一多，回傳的 LBR 封包可能是有順序或者是沒有順序的封包，因為乙太網路的封包不能夠確保所有回來的封包都是照順序的。這時便需要網路管理人員去分析對應的封包。

2.4 Linktrace Protocol

Linktrace Protocol 是由一個 MEP 上面的 MEP Linktrace Initiator 以及一個交換機上面的 Linktrace Responder 來實行的。其中所傳輸的 LTM(Linktrace Message)以及 LTR(Linktrace Reply)就是這個協定用來達成 Path Discovery 以及 Fault Isolation 的兩種訊息。

2.4.1 Linktrace Message，Linktrace Reply

LTM 亦是 multi-cast 的封包，裡面的 destination_address 參數也是由 802.1ag 標準依據 MD Level 所規範，LTM 封包會一直被交換機所傳送直到封包到達適當 MD Level 上的 MP 為止。這一個 MP 會中止 LTM 的傳輸，並且將 LTM 封包轉送到這個交換機的 Linktrace Responder 上面。

再者，Linktrace Responder 也會決定 MEP 或者 MIP 接收到 LTM 之後是否回應 LTR 封包，至於判斷的依據，在 802.1ag 標準裡面有詳述[1]。

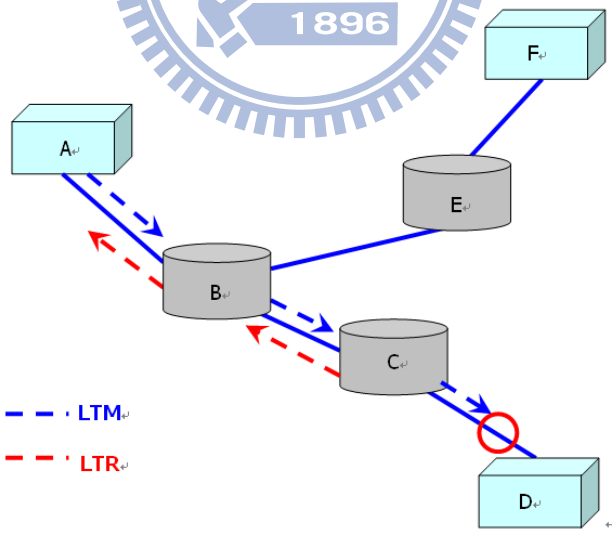

圖 2-4 Linktrace Protocol

我們用圖 2-4 來說明簡單的 Linktrace Protocol 運作方式。由 MEP A 製造 LTM 封包，讓 MIP B 以及 MIP C 都沿著路徑 B-C 傳送 LTR 回原點 MEP A，並朝著 MEP D 傳送 LTM 封包。故此協定可以達成路徑發現以及錯誤隔絕功能。

第三章

物件規劃、狀態圖以及事件處理

3.1 物件規劃

在我們的程式中，規劃了如下列的物件。

- Tester：為一個獨立的進程(Process)，裡面包含著接收指令的線程(Thread)。

  cliRcv：用來接收並處理管理人員指令的線程，為 CLI(Command Line Interface)的一部份。

- MEP：由 CLI 下指令後所創造的一個線程，擁有收送封包(sniff 程式)的能力，也是 802.1ag 協定中最重要的架構，裡面包含了 802.1ag 中定義的狀態機，如 MEP Continuity Check Receiver、MEP Continuity Check Initiator…等等。
- cliSend：為一個獨立的進程，此進程用意在於給予管理人員下達命令的介面。並藉由訊息佇列(Message Queue)和 cliRcv 溝通。

3.2 狀態圖

再這一小節，我們陳述如何將 802.1ag 標準上面的狀態圖轉換成最終我們所使用的程式碼。

3.2.1 MEP Continuity Check Receiver 在 802.1ag 標準內的表示

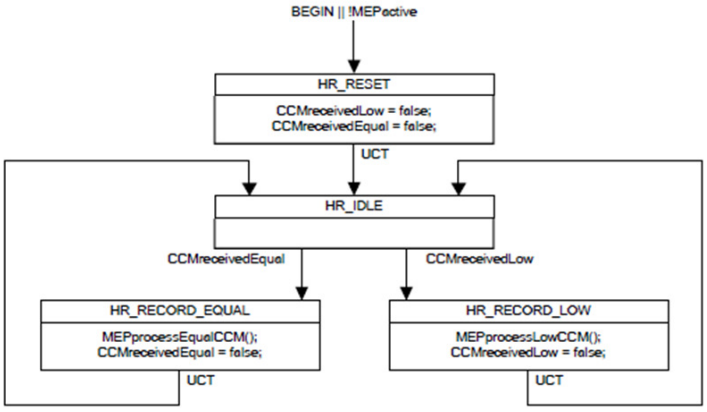

Figure 20-4—MEP Continuity Check Receiver state machine

圖 3-1 802.1ag 標準內的 MEP Continuity Check Receiver

圖 3-1 為 802.1ag 標準上面 MEP Continuity Check Receiver 的狀態圖，我們可以清楚看到裡面有四個狀態：HR_RESET、HR_IDLE、HR_RECORD_EQUAL 以及 HR_RECORD_LOW。

MEP Continuity Check Receiver 在被 MEP 創造出來時會先進入 HR_RESET 這個狀態，並且將兩個參數，CCMreceivedLow 以及 CCMreceivedEqual 設置成 false。接著狀態無條件的轉移(UCT)到 HR_IDLE，此時由於 CCMreceivedLow 以及 CCMreceivedEqual 都是 false，所以狀態機便在此停住，完成了初始的動作。等待有封包來時，Equal Opcode Demultiplexer 或者 Low Opcode Demultiplexer 的觸發。Equal(Low) Opcode Demultiplexer 的功能是將接收到的 CFM 封包分門別類，Equal 或者 Low 的差異在於接收到封包的 MD Level 與接收封包 MEP 的 MD Level 是相等或是較低。

假設 Equal(Low) Opcode Demultiplexer 判斷收到的 CFM 封包為 CCM，會先更改 CCMreceivedEqual(CCMreceivedLow)，之後再去觸發 MEP Continuity Check Receiver 的狀態圖，使得 MEP Continuity Check Receiver 的狀態會再從 HR_IDLE 進入 HR_RECORD_EQUAL(HR_RECORD_LOW)，做完了該狀態的事情後，又無條件的將狀態轉移到 HR_IDLE，等待下一次的觸發。

3.2.2 MEP Continuity Check Receiver 在 UML 工具上的表示

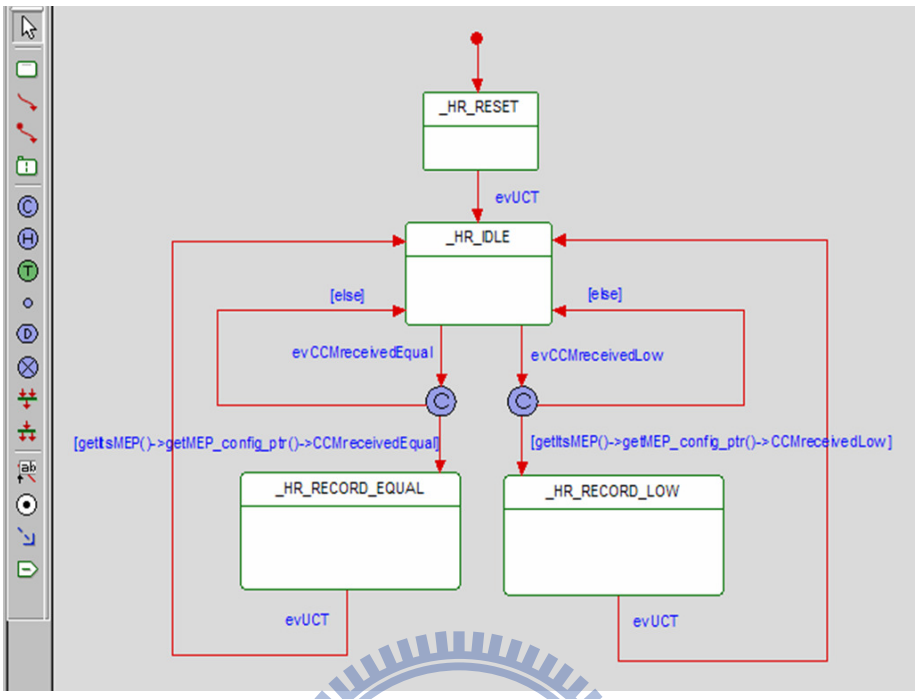

圖 3-2 UML 工具內 MEP Continuity Check Receiver 的表示

圖 3-2 為使用 UML 工具的表示方法，同樣的我們也做四個狀態，名稱與 802.1ag 標準內的相對應。對於標準內的 UCT 部份，我們在工具內可以自己產生給自己一個 evUCT 這樣的事件(使用 UML 工具中提供的 GEN()函式)，使狀態可以從_HR_RESET 轉移到_HR_IDLE。

而裡面的 evCCMreceivedEqual 或者 evCCMreceivedLow 皆是由 Equal(Low) Opcode Demultiplexer 對 MEP Continuity Check Receiver 來發送進而達到觸發的效果，可以達到與 802.1ag 標準內相同行為的狀態圖。

3.2.3 MEP Continuity Check Receiver 更改後的程式碼

最後我們將 UML 工具所產生出來的狀態圖做最後的轉換，取消了 UML 工具之中 Event 功能，大幅減少了程式碼所佔用的記憶體。並且將整個狀態圖用一個函式來概括，我們將其命名為 handle()。程式碼如下：

```
void MEPCCR::handle() {
        stateCount = 1;
        while ( stateCount )
        {
                switch ( myState ) {
                case _HR_RESET:
                //getItsMEP()->getMEP_config_ptr()->CCMreceivedLow=false;
                //getItsMEP()->getMEP_config_ptr()->CCMreceivedEqual=false;
                myState = _HR_IDLE;
                stateCount = 1;
                break;

                case _HR_IDLE:
                if ( getItsMEP()->getMEP_config_ptr()->CCMreceivedLow ) {
                                myState = _HR_RECORD_LOW;
                                stateCount = 1;
                }
                else if( getItsMEP()->getMEP_config_ptr()->CCMreceivedEqual ) {
                                myState = _HR_RECORD_EQUAL;
                                stateCount = 1;
                }
                else {
                                myState = _HR_IDLE;
                                stateCount = 0;
                }
```

```
            break;

                case _HR_RECORD_LOW:
                    MEPprocessLowCCM();
                    getItsMEP()->getMEP_config_ptr()->CCMreceivedLow = false;
                    myState = _HR_IDLE;
                    stateCount = 1;
                break;

                case _HR_RECORD_EQUAL:
                    MEPprocessEqualCCM();
                    getItsMEP()->getMEP_config_ptr()->CCMreceivedEqual = false;
                    myState = _HR_IDLE;
                    stateCount = 1;
                break;
            }
        }
    }
```

事實上，狀態圖本身就可以看成是一個 switch case 的集合，使用 C++語言之中的列舉(enum)，會讓整個函數看起來更加的有條理。

但是僅使用 switch case 是不夠的，為了達到與 802.1ag 標準相同行為的狀態圖，我們還加上了 while 迴圈以及一個 stateCount 參數，用來控制這個 handle() 函數何時停止，何時應該繼續運作，何時應該無條件的轉移狀態。

3.2.4 MEP Continuity Check Initiator 在 802.1ag 內的表示法

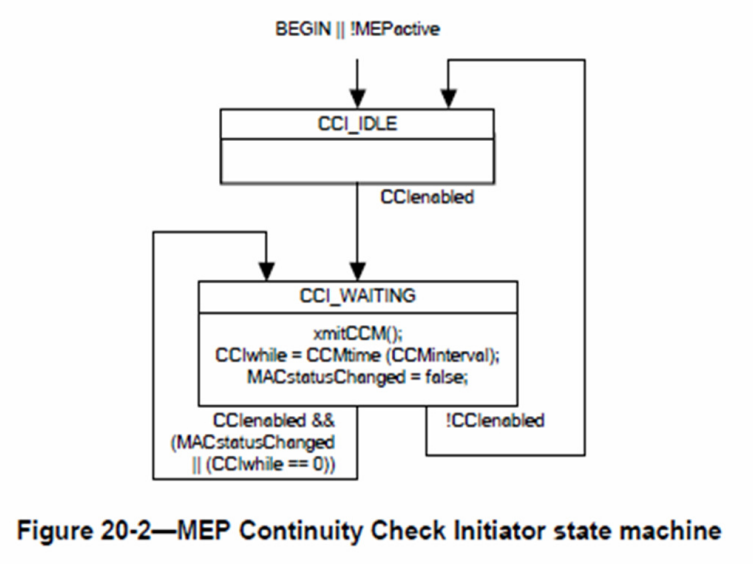

Figure 20-2—MEP Continuity Check Initiator state machine

圖 3-3 802.1ag 標準內的 MEP Continuity Check Initiator

圖 3-3 為 802.1ag 標準內的另一個狀態機，我們用此狀態機來說明如何處理有時間事件的狀況。

這個狀態機裡面只有兩個狀態，分別為 CCI_IDLE 以及 CCI_WAITING。首先進入到 CCI_IDLE，經由判斷 CCIenabled 為真或假的值，來決定是否進入 CCI_WAITING 這個狀態。CCIenabled 也是一個讓 MEP Continuity Check Initiator 可否傳送 CCM 封包的重要參數。凡進入 CCI_WAITING 狀態，第一件事就是傳送出一個 CCM 封包，由 xmitCCM()來達成，再來將 MACstatusChanged 設成 false，接著便是停留一段時間，這段停留的時間由 CCMtime()來做決定，不同的 CCMinterval 的設定對於 CCMtime()所造成的輸出都不相同。最後再判斷是否停留在 CCI_WAITING 狀態(繼續的週期性發送 CCM 封包)，亦或回到 CCI_IDLE 狀態(MEP Continuity Check Initiator 不工作)。值得注意的是，CCIenabled 是一個可以由網路管理人員經過 CLI(Command Lind Interface)更改的參數。

3.2.5 MEP Continuity Check Initiator 在 UML 工具上的表示

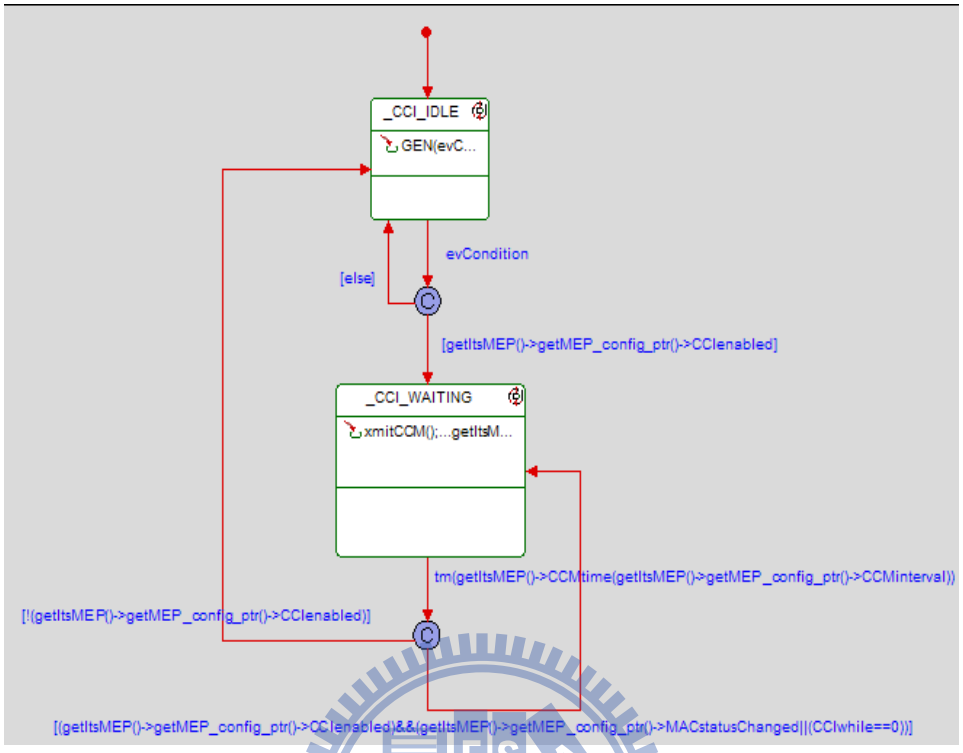

圖 3-4 UML 工具內 MEP Continuity Check Initiator 的表示

我們先將 802.1ag 標準裡的 MEP Continuity Check Initiator 做成如圖 3-4，一開始先進入_CCI_IDLE 之後，自動觸發(自己給予自己)事件 evCondition，先行判斷 CCIenabled 為真或假，才決定是否進入_CCI_WAITING。一旦進入_CCI_WAITING 狀態，行為便和 3.2.4 小節陳述的相同。唯一要注意的地方是，我們利用 UML 工具之中的 tm()函式來達到時間事件的處理，tm()這個函式就是倒數計時，等到時間消耗掉之後，便去判斷後面的條件式，來決定是否停留在_CCI_WAITING 或是回到_CCI_IDLE，只要 CCIenabled 持續的為 true(沒有被網路管理人員設定為 false)，那麼狀態機就會持續的保留在_CCI_WAITING 狀態，並且每隔一段時間(由 tm()這個函數倒數時間)，執行 xmitCCM()這個函式。

3.2.6 MEP Continuity Check Initiator 更改後的程式碼

這一部份我們結合了 mepScheduler 來幫助我們達到時間事件的處理，主要的函式便是 generateEvent()，我們要藉由這個函式，將狀態機與 mepScheduler 連接起來，mepScheduler 需要在裡面置有事件後，執行 run()函式才有意義，至於 generateEvent()詳細的觀念與作法我們在 3.3.2.1 節 Real-Time Scheduler 概念中會詳細闡述。

我們將 MEP Continuity Check Initiator 的程式碼完成如下：

```
void MEPCCI::handle(Event* ev_mepcci) {
    generateEvent();
    stateCount = 1;
    while ( stateCount ) {
        switch ( myState ) {
        case _CCI_IDLE:
        if ( getItsMEP()->getMEP_config_ptr()->CCIenabled ) {
            myState = _CCI_WAITING;
            stateCount = 1;
        }
        else
        {
            myState = _CCI_IDLE;
            stateCount = 0;
        }
        break;

        case _CCI_WAITING:
```

```
                xmitCCM();
                itsMEP->getMEP_config_ptr()->MACstatusChanged = true;
        if((getItsMEP()->getMEP_config_ptr()->CCMinterval==5)||(getItsMEP()->getMEP_config_ptr()
->CCMinterval==6) || (getItsMEP()->getMEP_config_ptr()->CCMinterval==7)) {
                getItsMEP()->getMEP_config_ptr()->MACstatusChanged = true;
            }
            else {
                ;
            }

            if(itsMEP->getMEP_config_ptr()->CCIenabled&&(   itsMEP->getMEP_config_ptr()->MA
            CstatusChanged||((getItsMEP()->getMEP_config_ptr()->CCIwhile)==0) ) ) {
                myState = _CCI_WAITING;
                    stateCount = 0;
            }
            else if ( !(itsMEP->getMEP_config_ptr()->CCIenabled) ) {
                myState = _CCI_IDLE;
                stateCount = 0;
            }
            else {
                ;
            }
            break;
        }
        }
}
```

這邊要注意，由於我們用 mepScheduler 取代了原本 UML 工具上的作法，所以 MEP Continuity Check Initiator 的狀態機也需要做些許的改變，來符合與 802.1ag 標準定義的行為。

我們首先在 MEP 的 handle()函式裡面先執行 MEP Continuity Check Initiator 的 generateEvent()函式，如下程式碼所示：

```
void MEP::handle() {
    addStateMachinesRelation();
    initialMEP_config(getMEP_config_ptr());
    setItsMepScheduler(p_mepScheduler);
    itsMepScheduler->getPRealSche()->reset();   // mepScheduler first reset();
    itsMEPCCI->generateEvent();
    itsMepScheduler->getPRealSche()->run();
}
```

這樣可以使第一個事件加到 mepScheduler 裡面。但是僅這樣子就執行 mepScheduler 的 run()函式時，只會讓 MEP Continuity Check Initiator 的 handle()被執行一次，而沒有辦法讓 MEP Continuity Check Initiator 持續被執行，原因在於呼叫一次 generateEvent()只會讓 mepScheduler 裡面多一個事件。

有鑑於此，我們要讓 MEP Continuity Check Initiator 持續被執行，就必須要在 MEP Continuity Check Initiator 的 handle()函式內加入 generateEvent()，使其每執行一次 handle()時一開始又再執行 generateEvent()，將事件加入到 mepScheduler 裡面，讓 mepScheduler 裡面持續有事件。如此我們將 MEP Continuity Check Initiaor 這台狀態機加到 mepScheduler 裡面就可以持續被執行。

3.3 事件處理

針對事件的處理，我們將其分為兩種類型，並且分別在實做上有不同的方法：

- 訊號事件(Signal Event)：

  自身觸發事件：802.1ag 標準內的 UCT(Unconditional Transition)即為此代表。

  一般訊號事件：需要經由其他狀態機來觸發的事件。

- 時間事件(Time Event)：

  每隔一段時間就去執行一次的事件：802.1ag 裡面的 MEP Continuity Check Initiator 每隔一個時間週期就會傳送 CCM 封包，此種事件謂之時間事件。

3.3.1 訊號事件

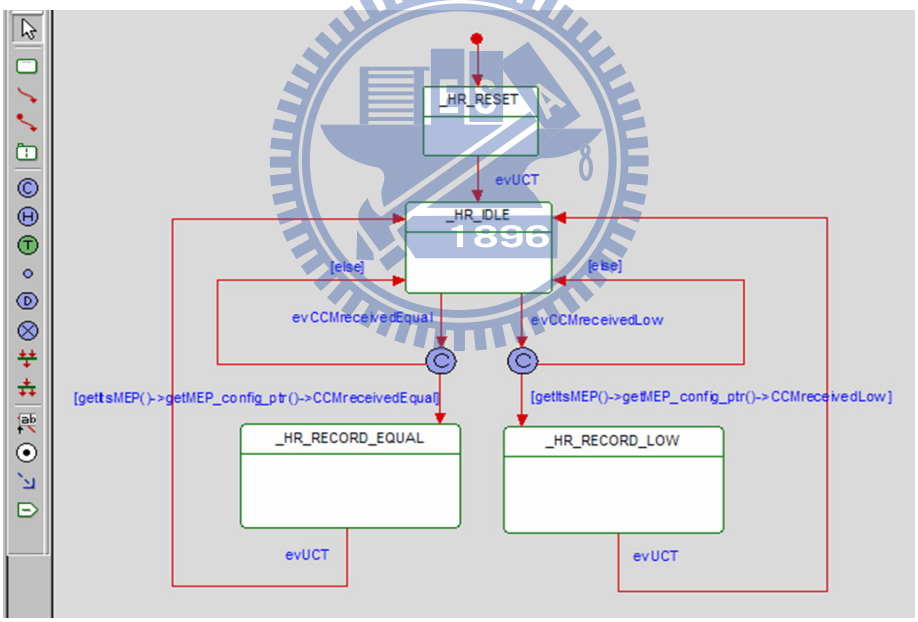

圖 3-5 訊號事件狀態圖說明

圖 3-5 為訊號事件的範例，裡面所示的 evUCT、evCCMreceivedEqual 以及 evCCMreceivedLow 都是屬於訊號事件。在 UML 工具中，對於 evUCT 這種無條件的狀態轉移，我們可以使用自身觸發事件，也就是進入狀態_HR_RESET 之後，自己產生一個事件 evUCT 使狀態轉換到_HR_IDLE，要使用這樣的方法(自身觸發)原因在於每個狀態都不應該平白無故的轉移，必須要有觸發(trigger)才會有狀

態轉移。而在我們最後所使用的程式碼中，是使用 while loop 與 stateCount 參數來達成，或者不需要一個明顯的狀態來處理這樣的情況。

再者，對於 evCCMreceivedEqual 或者 evCCMreceivedLow 這兩個事件，並不是由自己所造成，在 802.1ag 標準之中，造成狀態由_HR_IDLE 轉換到 _HR_RECORD_EQUAL 或者 _HR_RECORD_LOW。是由 Equal Opcode Demultiplexer 或者 Low Opcode Demultiplexer 來觸發，所以理所當然的要由 Equal / Low Opcode Demultiplexer 產生事件給 MEP Continuity Check Receiver，觸發狀態轉移。值得一提的是，在 UML 工具中有提供 GEN()這樣的函式可供使用來產生事件，不論是自身觸發或是由其他的狀態機產生皆可使用 GEN()函式。

最後在我們更改後的程式碼內，對於 evUCT 的做法其實相當簡單，由下列節錄的程式碼就可以輕易看出：

```
stateCount = 1;
while ( stateCount )
{
    switch ( myState ) {
    case _HR_RESET:
    //getItsMEP()->getMEP_config_ptr()->CCMreceivedLow=false;
    //getItsMEP()->getMEP_config_ptr()->CCMreceivedEqual=false;
    myState = _HR_IDLE;
    stateCount = 1;
    break;
    case _HR_IDLE:
    if ( getItsMEP()->getMEP_config_ptr()->CCMreceivedLow ) {
                myState = _HR_RECORD_LOW;
                stateCount = 1;
    }
```

由_HR_RESET 狀態到_HR_IDLE 狀態就是 evUCT 的效果，只要搭配 while loop 以及控制的 stateCount 就可以輕易的做到這件事情，讓狀態無條件的轉移。

另外對於 evCCMreceivedEqual 或者 evCCMreceivedLow 這兩種由其他狀態機觸發的事件，我們使用的方法也相當容易了解，在 3.2.3 我們有提到 MEP Continuity Check Receiver 的程式碼，裡面可以清楚看到，系統初始化時，CCMreceivedEqual 以及 CCMreceivedLow 皆為 false，所以狀態機會停在_HR_IDLE 的狀態並且設定 stateCount 為 0，被閒置在此狀態。當一個 CFM 封包被 Equal Opcode Demultiplexer 給判斷為 CCM 形式同時，Equal Opcode Demultiplexer 會再度呼叫 MEP Continuity Check Receiver 的狀態機，也就是再一次的呼叫 MEP Continuity Check Receiver 的 handle()函式，此動作便是我們處理 evCCMreceivedEqual 或者 evCCMreceivedLow 的方法，簡單來說就是讓前一個狀態機(Equal Opcode Demultiplexer)呼叫現在這個狀態機(MEP Continuity Check Receiver)的函式(handle())，就可以達成此種類型的訊號事件。

3.3.2 時間事件

對於時間事件的處理，是一個非常重要的部份，我們使用 Real-Time Scheduler 來針對時間事件做一系列的處理動作。在我們的程式碼裡面的 Real-Time Scheduler 稱為 mepScheduler。

不使用 C 語言提供的 sleep()函式原因主要在於 multi-thread 狀況下使用 sleep()可能會被搶走 CPU 的使用權而使狀態機的運行中斷而使行為與 802.1ag 標準內規定的不符。而擁有越多時間事件，一般狀況都是在一個進程(Process)內增加線程(Thread)，這樣會相當耗費資源，使用 Scheduler 可以避免這樣的狀況產生。

3.3.2.1 Real-Time Scheduler 的概念

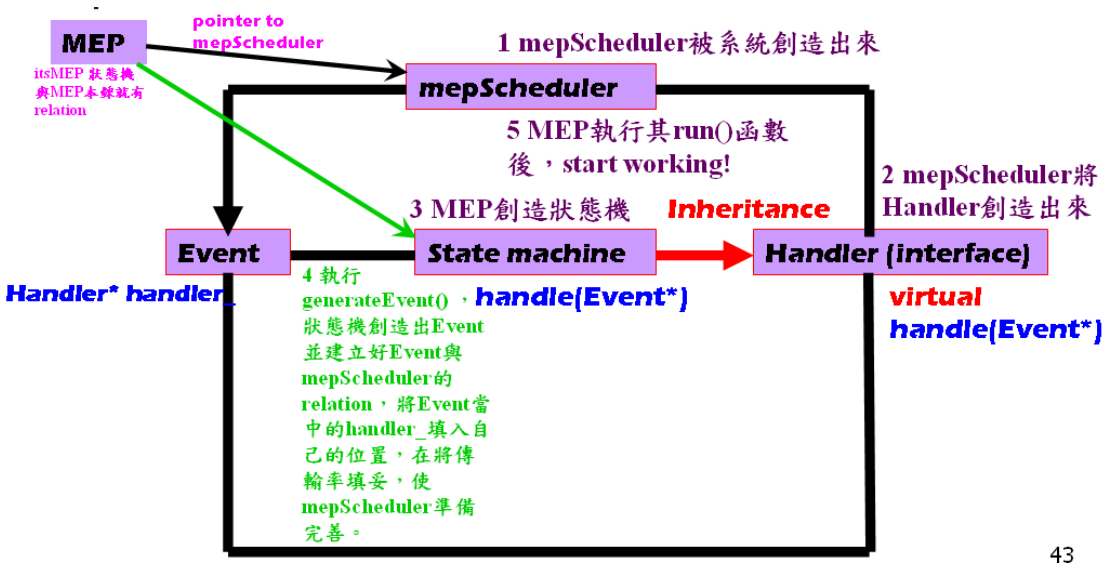

圖 3-6 Real-Time Scheduler 架構圖

圖 3-6 我們用來說明 mepScheduler 在運作前需要建立起來的物件關係圖。整個 mepScheduler 之中最基本的 class 就是 Handler 以及 Event。

mepScheduler 這個 class 裡面主要的函式為 dispatch()、run()、reset()以及 schedule()。

- reset()：將 mepScheduler 初始化的函數，主要是將 clock 初始化。
- schedule()：狀態機將 Event 的指標、Event 當中 handler_的指標位置(其實就是指到狀態機 handle()的位置)以及每個事件的延遲時間(Delay Time)給予 mepScheduler 的函數。
- run()：最後讓整個 mepScheduler 開始運作的函式。
- dispatch()：mepScheduler 去執行當下事件所用的函式，也就是不斷的去抓 handler_的位置來執行。

Handler 是對於所有事件最基本的 class，裡面擁有 handle()這個函數，型別為 virtual，代表的就是要執行的時間事件，也就是擁有時間事件狀態機的 handle()函式，使用 virtual 型別是為了給狀態機的 handle()函式繼承使用。

Event 這個 class 裡面擁有型別為 Handler 的 handler_指標，用意在於指到現

在要執行的時間事件(其實就是擁有時間事件狀態機 handle()函式)的位置。

當一個事件的排程時間到達，這個事件就需要被傳到 handler_指標上面去，然後利用 Handler 之中的 handle()函式，也就是指到狀態機的 handle()函式將事件消耗掉。

關於繼承的使用原因，在於本身 mepScheduler 的 Handler 已經寫好了對於 mepScheduler 以及 Event 之間的關係，如果不使用繼承，那只要有時間事件的狀態機都要去寫一個和 Handler 架構一樣的框架來使用，實在是相當不方便。所以選擇使用繼承方式，讓各個擁有時間事件的狀態機都可以使用 Handler 框架，更加的方便。

再者我們要說明狀態機向 mepScheduler 註冊的 generateEvent()函式，即是圖 3-6 的步驟 4：

```
void MEPCCI::generateEvent() {
    Event* anEvent = new Event();
    anEvent->setHandler_(this);
    itsMEP->getItsMepScheduler()->getPRealSche()->schedule(anEvent->getHandler_(), anEvent, /*getItsMEP()->getCCMinterval()*/0.5);
}
```

由於狀態機有時間事件，故一開始必須將 Event 的物件給創造出來，並且將 Event 物件中的 handler_指向狀態機的 handle()函式，故使用 this 指標。最後再向 mepScheduler 註冊，將 Event 的指標，Event 當中 handler_的位置以及時間事件的 Delay Time 給予 mepScheduler，完成整個動作。

3.3.2.2 Scheduling Scenario

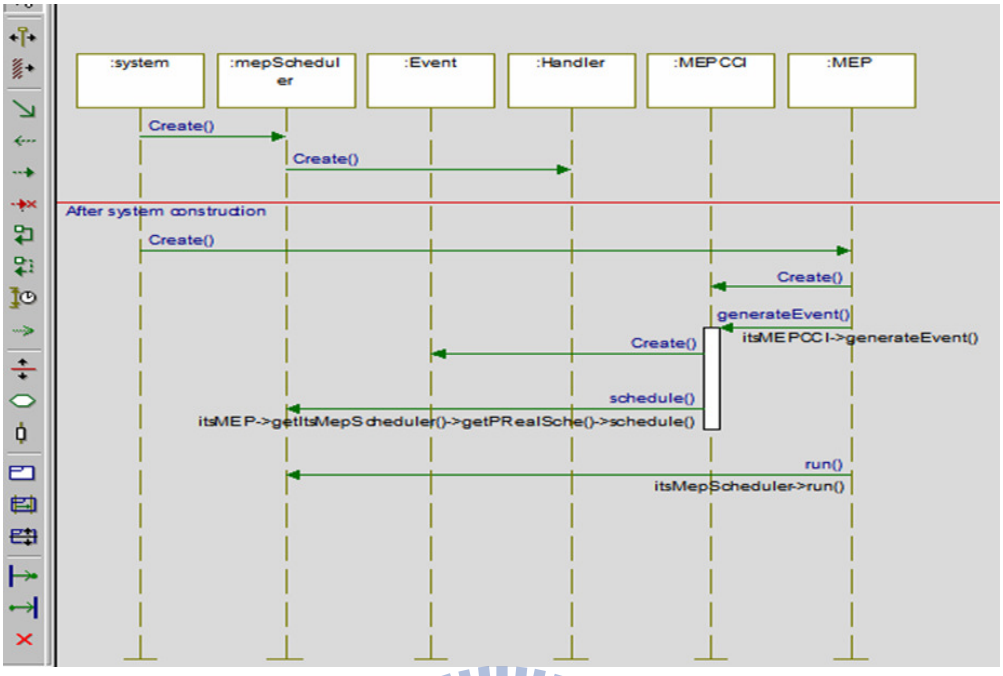

圖 3-7 mepScheduler Scenario

由圖 3-7 我們更可以了解到 mepScheduler 的運作流程。首先當系統一開始初始化時，就會先將 mepScheduler 創造出來，mepScheduler 也緊接著將 Handler 這個物件創造出來。接著當系統收到由 cliSend 下達的指令將 MEP 這個線程(Thread)創造出來時，MEP 本身又會將裡面所含得狀態機一個一個創造出來。又因為 MEP Continuity Check Initiator 是一個擁有時間事件的狀態機，所以 MEP 接著需要去執行 MEP Continuity Check Initiator 的 generateEvent()函式，方可將其與 mepScheduler 連接上關係，接著就可以執行 mepScheduler 裡面的 run()函式。整個 mepScheduler 就開始運作。

第四章

實驗測試架構與參數設定

本章節敘述我們針對量測 100 個 CCM 封包處理時間所做的實驗架構、測試方法以及對於狀態機裡面的參數設定。

- 測試使用儀器：

  筆記型電腦 1：

  作業系統：Microsoft Windows XP Processional Version 2002。

  中央處理單元：Intel(R) Pentium(R) M processor 1.60GHz。

  VMWare：Linux Fedora 8。

  VMWare 記憶體：256MB。

  筆記型電腦 2：

  作業系統：Microsoft Windows XP Processional Version 2002。

  中央處理單元：Intel(R) Pentium(R) M processor 1.60GHz。

  VMware：Linux Fedora 8。

  VMware 記憶體：256MB。

- 實驗架構：

  由於僅測試 CCM 封包處理時間，所以我們設置了最簡單的一個架構。我們利用 ccmSocketRaw 這個程式可以對 MEP 連續送出 50 個封包，我們量測 MEP 裡面的 MEPCCR 處理時間，如圖 4-1 所示。

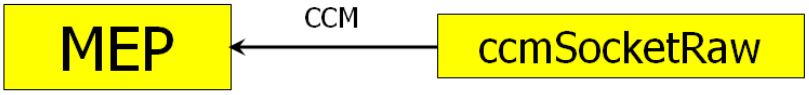

圖 4-1 實驗測試架構

我們使用兩台筆記型電腦，MEP 所在的為筆記型電腦 1，而 ccmSocketRaw 為筆記型電腦 2。兩台電腦網路線互相接妥後，即可測試。

- 測試方法：

  兩台筆記型電腦備妥後，皆利用 VMware 開啟 Fedora 8 的系統。

  筆記型電腦 1 利用 putty 連接到 VMWare 內，開啟兩個 process，分別作為 cliSend 以及 MEP 用。輸入./cliSend 指令，可將 cliSend 這個 process 創造出來。同時另一個 process 輸入./procotolKernel 後，即可將 protocolKernel 創造出來，並且等待 cliSend 下達開啟 MEP 的指令。

圖 4-2 筆記型電腦 1 操作介面

圖 4-2 裡面有兩個獨立的 process，其一為 cliSend，另一個則是 protocolKernel。

圖 4-3 執行./protocolKernel 指令後的系統，等待 cliSend 輸入開創 MEP 的指令(在筆記型電腦 1)

圖 4-4 下達 createMEP 指令的 cliSend process(在筆記型電腦 1)

```
1
msqid_rcv : 0
msqid_snd : 32769
****** cliRcv ******
Managed object that want to be modified : createMEP
cmd : createMEP
receivedValue : 1
************ MEPMainFunction has been executed ****************
-------------------- addStateMachinesRelation() --------------------------
===
===
===
===
===
===
===
===
===
 MEPLTI initRelation()
===
===
===
===
===
insert() in SequenceLR_AMM has been executed.
Head=165572976.
insert() in SequenceLR_MEPLTI has been executed.
Head=165572944.
===
-------------------- addStateMachinesRelation() finished ------------------------
handle() in MEP has been executed.
clock_ = t.
eth3 is up!
Promiscuous Mode
Corresponding message has been received.
****** cliRcv ******
```

圖 4-5 接續圖 4-3，系統接收到 createMEP 指令後開創出準備接收封包的 MEP(在筆記型電腦 1)

```
root@polarbear:/home/onu/ONU/1104Apps/protocolKernel
root@polarbear protocolKernel]# ./ccmSocketRaw
```

圖 4-6 在筆記型電腦 2 下達./ccmSocketRaw 指令，會送出 50 個封包

圖 4-7 送完 50 個封包，封包個數從 0 至 50(在筆記型電腦 2)

我們需要輸入兩次的./ccmSocketRaw 以便送出 100 個封包並測量其時間。

```
root@polarbear:/home/can/ONU/1104App/protocolKernel
insert() in SequenceLR_MEPLTI has been executed.
Head=152392016.
===
-------------------- addStateMachinesRelation() finished --------------------------
handle() in MEP has been executed.
clock_ = t.
eth3 is up!
Promiscuous Mode
Corresponding message has been received.
****** cliRcv ******
start time tv.tv_usec = 121055
end time tv2.tv_usec = 121325
start time tv.tv_usec = 123838
end time tv2.tv_usec = 124406
start time tv.tv_usec = 127293
end time tv2.tv_usec = 128170
start time tv.tv_usec = 141672
end time tv2.tv_usec = 142296
start time tv.tv_usec = 155764
end time tv2.tv_usec = 156286
start time tv.tv_usec = 156468
end time tv2.tv_usec = 156468
start time tv.tv_usec = 158846
end time tv2.tv_usec = 158862
start time tv.tv_usec = 160157
end time tv2.tv_usec = 160543
start time tv.tv_usec = 160556
end time tv2.tv_usec = 161858
start time tv.tv_usec = 187340
end time tv2.tv_usec = 187373
start time tv.tv_usec = 187387
end time tv2.tv_usec = 187404
start time tv.tv_usec = 187414
end time tv2.tv_usec = 187430
start time tv.tv_usec = 187441
end time tv2.tv_usec = 187457
start time tv.tv_usec = 187467
end time tv2.tv_usec = 187483
start time tv.tv_usec = 187494
end time tv2.tv_usec = 187510
start time tv.tv_usec = 187520
end time tv2.tv_usec = 187536
start time tv.tv_usec = 187547
end time tv2.tv_usec = 187563
start time tv.tv_usec = 187573
end time tv2.tv_usec = 187589
```

圖 4-8 MEP 接收到封包後印出時間(在筆記型電腦 1)

以上圖 4-2 至圖 4-8 為我們測試的流程以及所需要輸入的指令。

- 參數設定：

針對我們的實驗環境，我們將重要且會影響實驗的參數做一些設定並且列出。事實上，裡面還有其他的參數，但是並不會對於我們的實驗架構及結果造成影響，故不在此一一陳述。

ccmSocketRaw 設定：

MD Level 設定為 6(自己定義，範圍由 0~7 都可以)。

Opcode 設定為 1，此為 CCM 封包所應該帶的 Opcode。

接收端 MEP 設定：

MD Level 設定為 6，配合 CCM 封包的 MD Level。

第五章

實驗結果

我們偵測時間的方法是利用 C 語言所提供的 gettimeofday()函式。

struct timeval tv;

int time;

gettimeofday(&tv, NULL);

printf(" time tv.tv_usec = %d\n", tv.tv_usec);

利用上面的程式碼，只要在收到封包的瞬間測量一次時間，以及 MEP Continuity Check Receiver 處理完封包的瞬間再測量一次時間即可。

最後是我們所量測到處理 CCM 封包的時間，圖表中數值代表的是 count 數目，使用 gettimeofday()函式比例為每 10^6 個 count 值代表 1 秒，我們將時間單位置為 μsec，此時 count 數目就會等於處理封包的秒數。除此之外，我們將樣本數設置為 100 個，將數據整理為以下圖表。

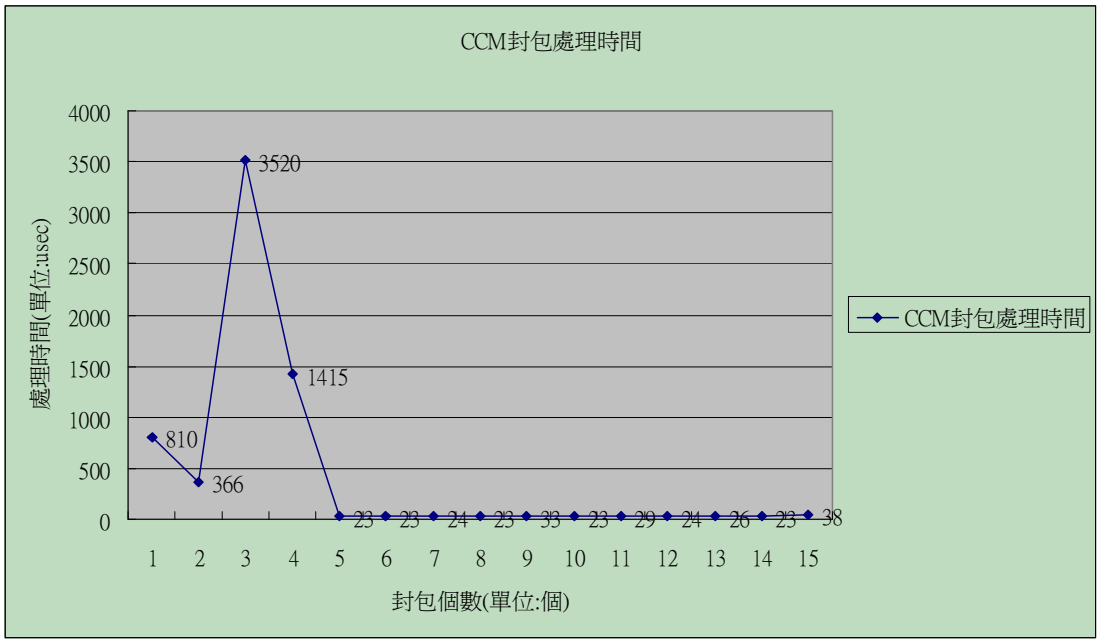

圖 5-1 第 1 個至 15 個封包處理時間

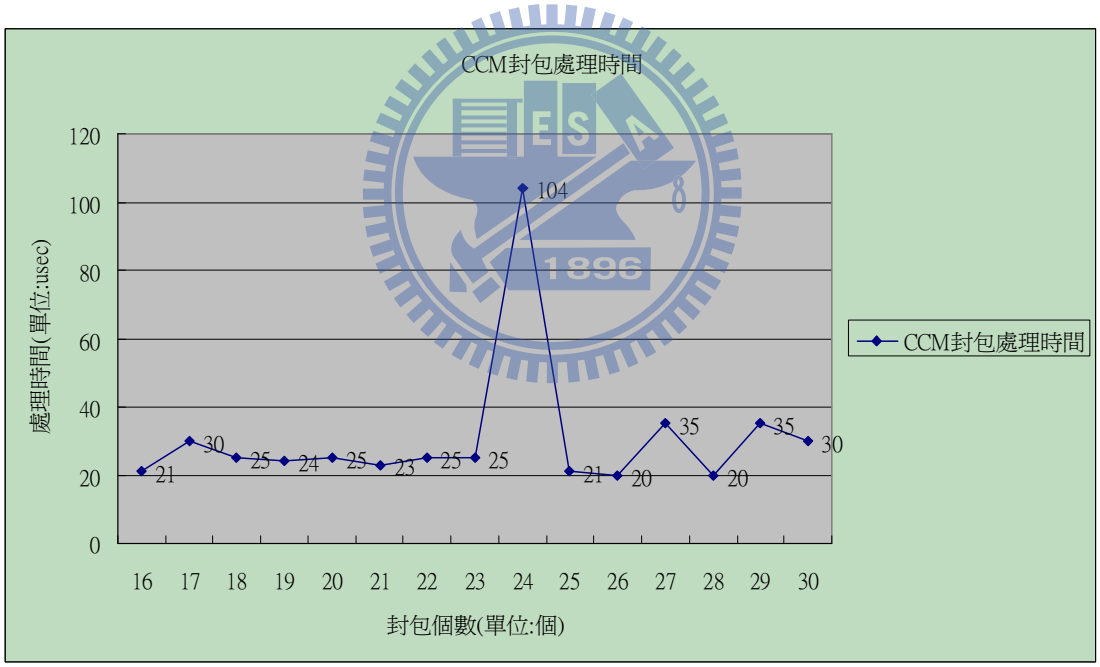

圖 5-2 第 16 個至 30 個封包處理時間

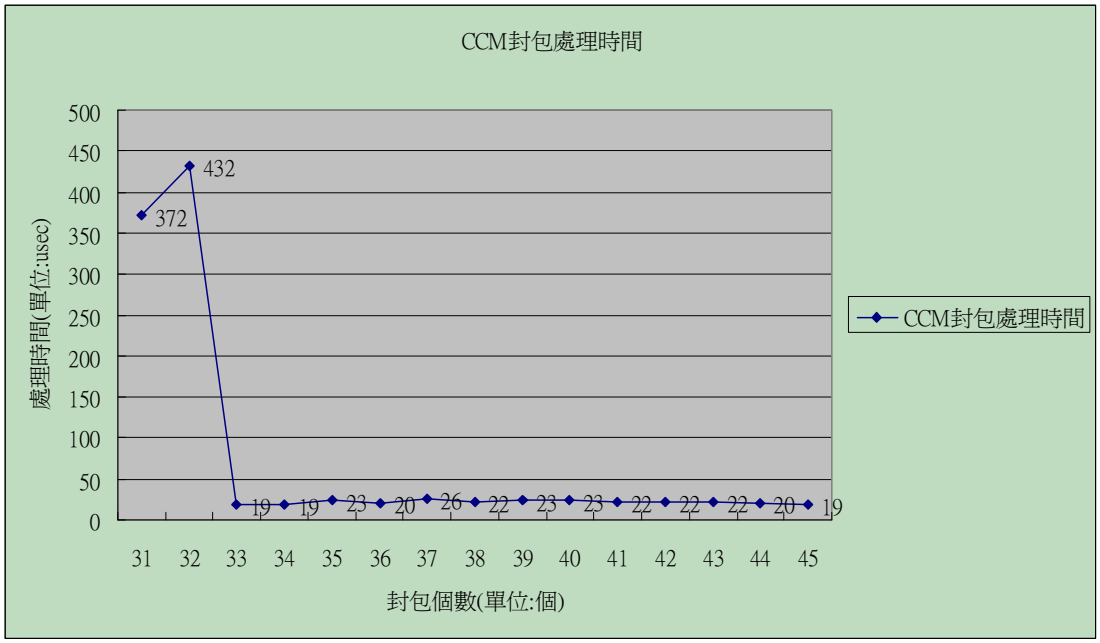

圖 5-3 第 31 個至 45 個封包處理時間

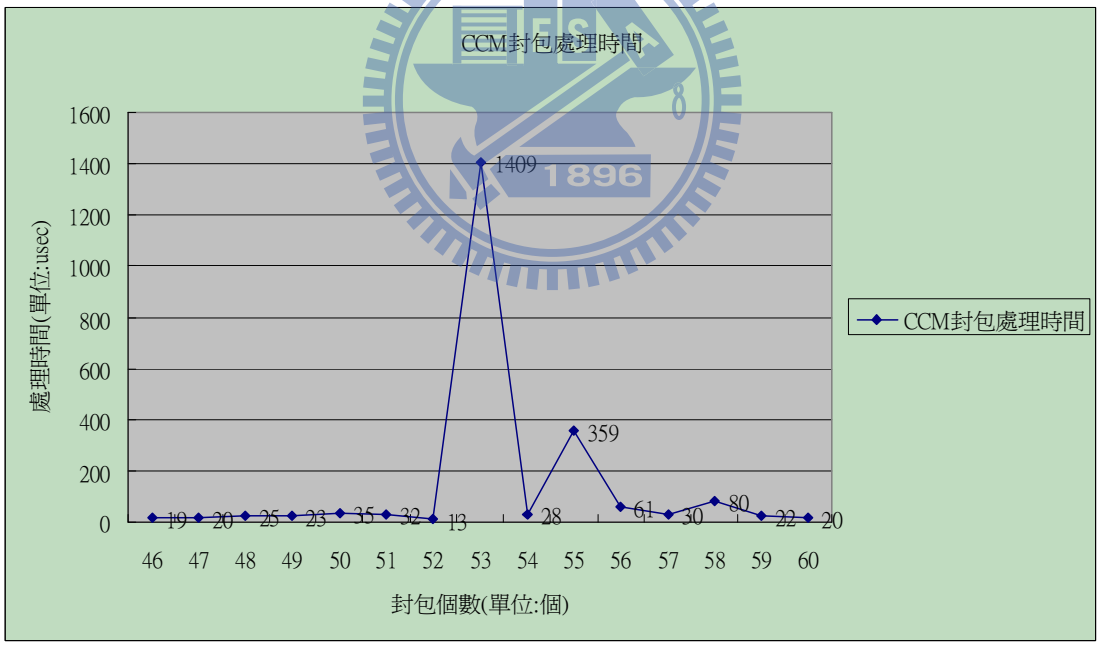

圖 5-4 第 46 個至 60 個封包處理時間

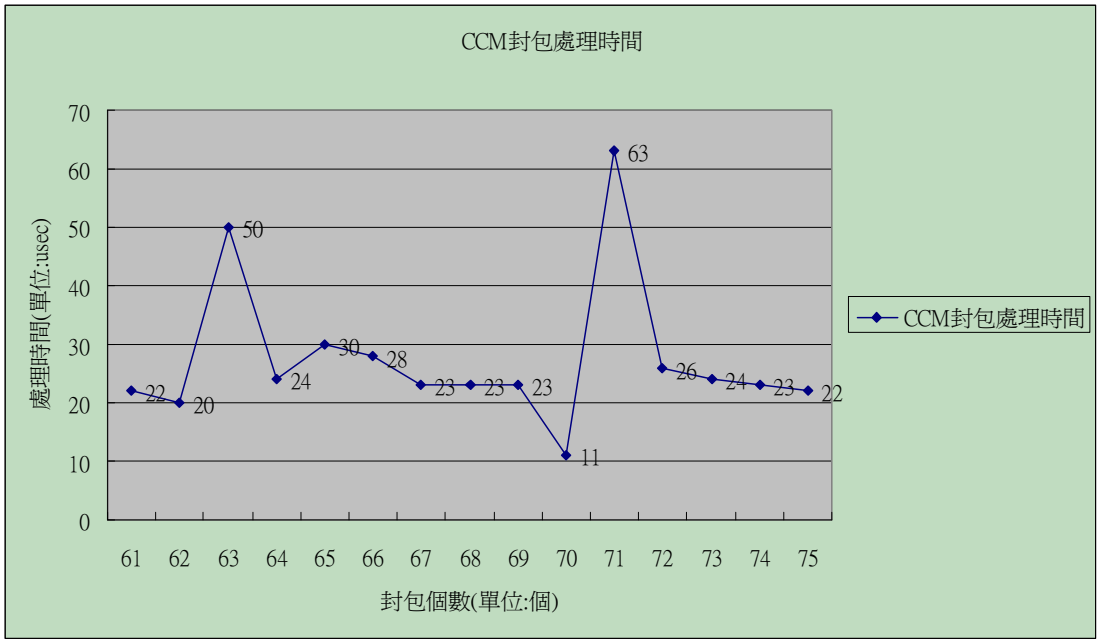

圖 5-5 第 61 個至 75 個封包處理時間

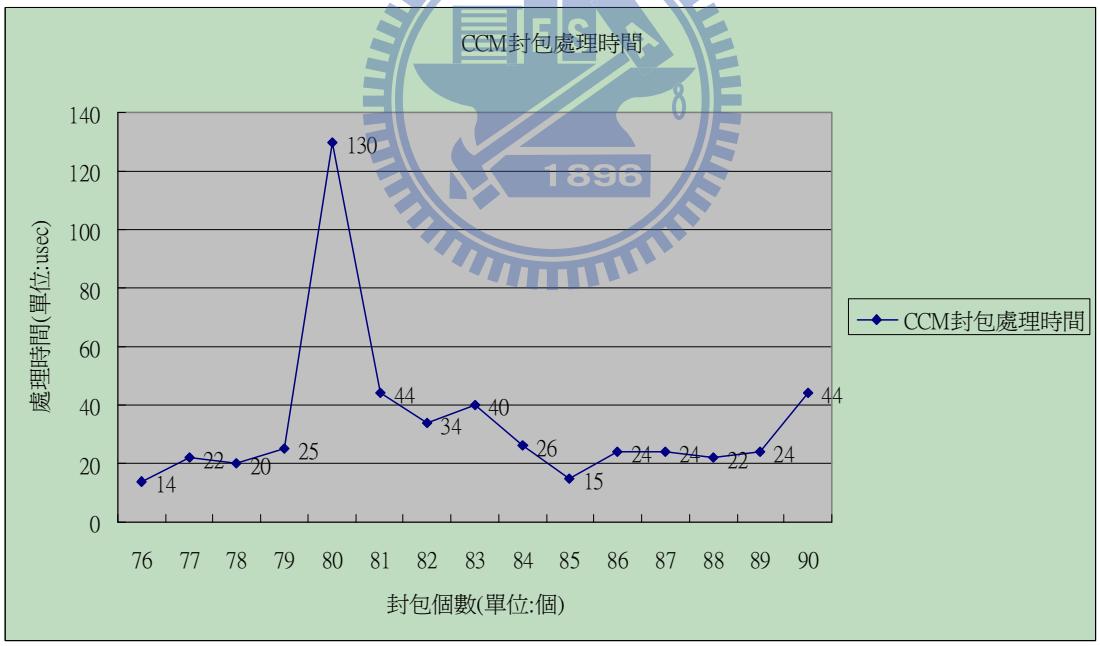

圖 5-6 第 76 個至 90 個封包處理時間

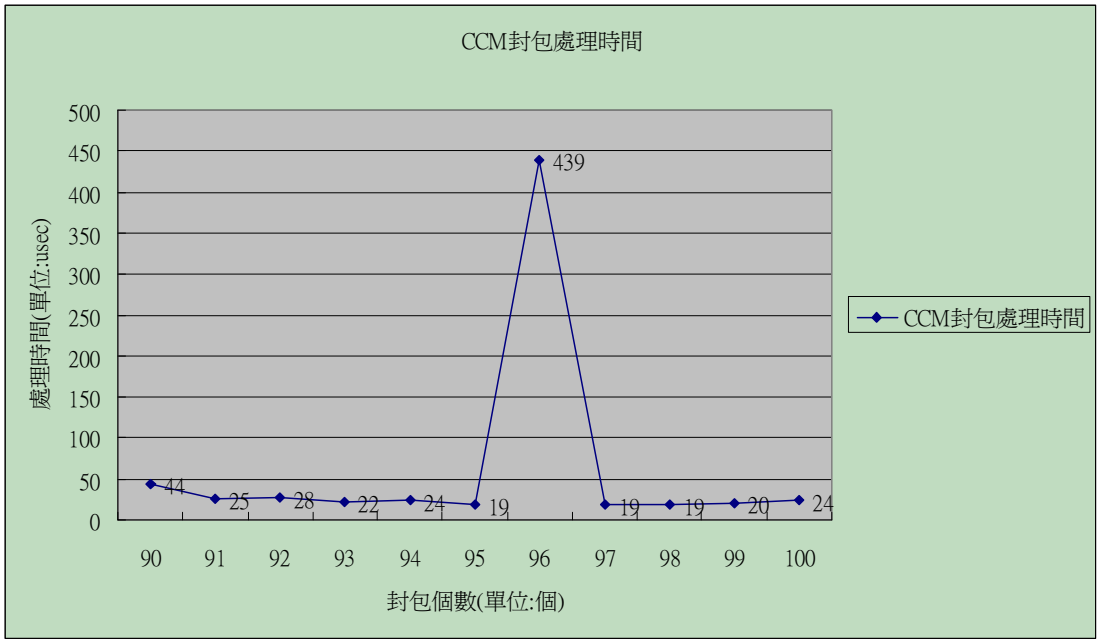

圖 5-7 第 91 個至 100 個封包處理時間

處理時間 / 封包統計個數	
10 至 20μsec	11
20 至 30μsec	64
30 至 40μsec	8
40 至 50μsec	3
50 至 60μsec	0
60 至 70μsec	2
70 至 80μsec	1
80 至 90μsec	0
100μsec 以上	11

圖 5-8 統整處理時間

第六章

結論

我們所使用的 Scenario-Driven 流程，與以往的 Event-Driven 比較，最大的優勢是可以直接將 UML 工具所產生出來的 Scenario(也就是程式運行的 Sequence Diagram)用來與所開發的協定交互驗證。而本論文的主軸在於發展一套以 UML 觀念來製作有 Real-Time 要求而且複雜 protocol 的好方法。

我們利用第五章的數據圖表來驗證使用 Scenario-Driven 這套方法在電腦上運行的可行性及穩定性。我們可以看出：實驗量測時間集中在 10μsec 至 30μsec，時間集中性可以代表著 Scenario-Driven 的做法在電腦開發層面以及運行層面是相當穩定的。藉以驗證 Scenario-Driven 是可以用在電腦上開發的可靠方法。對於極少數的處理時間會提升到數百亦或數千個 μsec 等級，只需要取多次的平均值即可消除這項問題，另外，如果需要在一個規範的時間內偵測出錯誤，除了 CCM 的傳輸週期需要更改之外，對於所使用的儀器也需要更高的等級才可配合偵錯，有關於這方面的研究，將在未來深入探討。

再者，Scenario-Driven 這種方法其實並不限制於使用在 UML 的工具上。它是一系列的開發程式的流程以及觀念。事實上，排除了 UML 工具的輔助，利用這種 UML 思維以及 Scenario-Driven 的流程，對於開發大型程式亦有相當大的幫助，不論在提昇整體規劃或是程式完成速度方面，未來我們將加入 MIB 的功能以及 Scenario 與 802.1ag 協定之間的測試，讓這套流程能夠得到更進一步的驗證及提升。

參考文獻

[1] "IEEE Standard for Local and Metropolitan Area Networks – Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management," 2007.

[2] M. McFarland, S. Salam and R. Checker, "Ethernet OAM: key enabler for carrier class metro ethernet services," *IEEE Communications Magazine*, vol. 43, no. 11, Nov. 2005.

[3] P. Reddy and S. Lisle, "Ethernet aggregation and transport infrastructure OAM and protection issues," *IEEE Communications Magazine*, vol. 47, no. 2, Feb. 2009.

[4] Y.-X. Li, Lei He and Yu-Zhen Li, "Telecom Ethernet OAM Research in the Metropolitans Area Network Multi-Operation Platform", *Apperceiving Computing and Intelligence Analysis, 2008. ICACIA 2008. International Conference on*, 13-15 Dec. 2008.

[5] Strembeck, M., "Scenario-Driven Role Engineering", *Security & Privacy, IEEE*, vol. 8, Jan.-Feb. 2010.