國立交通大學

資訊科學與工程研究所

碩士論文

設計與實作一個網路模擬雲端系統

Design and Implementation of a Network Simulation Cloud

研究生：蔡佳玟

指導教授：王協源　教授

中華民國一百零二年一月

設計與實作一個網路模擬雲端系統

Design and Implementation of a Network Simulation Cloud

研 究 生：蔡佳玟　　Student：Chia-Wen Tsai

指導教授：王協源　　Advisor：Shie-Yuan Wang

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Computer Science

December 2012

Hsinchu, Taiwan, Republic of China

中華民國一百零二年一月

設計與實作一個網路模擬雲端系統

學生：蔡佳玟　　指導教授：王協源　教授

國立交通大學 資訊科學與工程研究所 碩士班

摘　　要

透過使用網路模擬器，網路協定研究人員可不受時間、空間及設備限制來研究複雜的網路。研究人員通常需花費許多時間進行網路模擬實驗，才得以取得較可信的結果。隨著雲端運算的發展，雲端使用者可輕易地取得較多的運算能力。我們相信使用雲端的運算資源可減少網路模擬所需的時間。

在本論文中，我們以 EstiNet 網路模擬器及 Openstack 雲端平台為基礎，設計與實作一網路模擬雲端系統。藉由使用此網路模擬雲端系統，研究者可節省許多花費於模擬實驗的時間。我們也測量此系統執行網路模擬的效能，結果顯示此雲端系統可減少多模擬工作的總時間。

關鍵字：網路模擬、雲端計算

Design and Implementation of a Network Simulation Cloud

Student: Chia-Wen Tsai Advisor: Shie-Yuan Wang

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

With a network simulator, a network protocol researcher can study a complex network without limitations of time, space and device. A researcher usually spends much time to obtain simulation results from many network simulations. With cloud computing, a cloud user can easily obtain more computing power. We believe that using computing power in cloud will reduce time spent on network simulations.

In this thesis, we design and implement a network simulation cloud on the EstiNet network simulator and Openstack cloud platform. By using this network simulation cloud, a researcher saves much time spent on simulations. We also evaluate performance of this network simulation cloud. The result shows that this network simulation cloud reduces the total time spent on simulations.

Keyword: Network Simulation, Cloud Computing

誌　　謝

首先，我要感謝指導老師王協源老師在學術上、論文撰寫及口頭報告上許多的指導，讓我在系統的設計與實作上有許多學習，並能順利完成碩士論文與口試。感謝吳曉光老師、李端興老師以及周承復老師擔任我的口試委員，給予我許多寶貴的建議。

平時與 NSL 的學長姐、同學以及學弟妹的聊天哈拉以及認真討論，讓我擁有相當充實歡樂又溫馨的研究生生活，謝謝你們的支持與鼓勵。其中要特別感謝都都跟韻立在計畫上的付出及配合，還有體諒我這不太會當頭的頭。也要感謝 Bobo 學長在我遇到問題時給予相當多的協助。

感謝可愛的室友，小乖、野口跟阿璞忍受我這個高興時就退化成小朋友、煩躁時就一直碎碎念的幼稚室友。很高興可以跟野口、小乖當五年半的室友，謝謝兩位聽我說一拖拉庫的話，讓我低落、煩亂的心情得以平復。

謝謝國高中的朋友們透過網路給予我的支持。我想對某個笨蛋說：「嘿，我拿到碩士了，我們會連妳的份一起好好活下去，去看妳無緣見到的世界。」

再來，我要感謝我的家人。謝謝爸媽的養育跟照顧，讓我可以讀自己喜歡的科系，能夠無後顧之憂的取得學位。謝謝老弟在我寫碩論期間忍受我極端幼稚的行徑。此外，我想將這篇論文，獻給在天上的奶奶。

最後，我想感謝歆昀老師，謝謝她這一年半的陪伴。

研究所這段時間，無論在學術跟知識上還是在生活上，對我是相當特別而且有許多成長的一段時間，謝謝也祝福所有出現在我生命中的人們！

蔡佳玟 2013.1

目　　錄

圖　目　錄

表　目　錄

第1章 緒論

1.1 研究動機

隨著科技進步，各界不斷研發新的網路協定及技術。在網路協定的研究中，研究者需透過實驗測試其設計的協定是否能正常運作、取得數據以了解傳輸效能。研究者可以真實硬體設備或網路模擬進行相關網路實驗。相較於以硬體設備進行的網路實驗，網路模擬實驗可讓研究者以相對較少的資源得到較大的彈性，如規模較大的網路拓撲或尚未有硬體設備的網路協定。

網路模擬實驗需要大量運算資源。對於一個實驗，研究者需要執行多次模擬才能取得較為可信的結果。另外，對於一個網路拓撲，研究者可能需要修改拓撲中不同的網路參數，以觀察網路協定的運作。因此，要取得一個網路實驗的完整數據，以單一機器與人力需耗費大量的時間。

在本論文中，我們希望以雲端系統眾多機器的運算能力及自動化處理降低研究者執行模擬實驗所需要時間。我們以 EstiNet 網路模擬器為基礎，結合 Openstack 平台，設計、實作並建置一網路模擬雲端系統的雛形，提供使用者網路模擬的應用服務。完成設計、實作與建置後，我們在此雲端系統上執行網路模擬實驗，觀察並分析此雲端系統的效能。

1.2 章節概述

本篇論文的章節組成如下，第二章將介紹 EstiNet 網路模擬器及 Openstack，讓讀者了解我們開發此雲端系統的平台及基礎。第三章說明我們的系統架構，分別介紹軟體及硬體架構。第四章說明我們的設計目標、設計方式以及實作細節。第五章先介紹效能測量指標以及環境，接著說明我們在此平台上進行的效能測量及分析。第六章是在設計及實作此系統中，我們發現未來可加強的地方。第七章則為此篇論文的結論。

第2章 背景

2.1 EstiNet 網路模擬器介紹

在本論文中，我們以思銳科技公司所開發的 EstiNet 網路模擬器作為雲端系統中運行網路模擬工作的模擬器。本節將介紹 EstiNet 網路模擬器與雲端系統有關的特性、元件及架構。

EstiNet 網路模擬器前身為 NCTUns，是一擁有高真實性及擴充性的網路模擬器，可模擬多種有線、無線網路設備及網路協定。使用者可於 EstiNet 網路模擬器中運行真實世界的應用程式，使用者不需要特別為 EstiNet 網路模擬器的環境開發特殊的應用程式製造網路流量、進行傳輸。

EstiNet 網路模擬器使用分散式架構，讓多名使用者可同時使用多台電腦進行網路模擬。EstiNet 模擬器的分散式架構如圖 2-1 所示，此架構中有 GUI（Graphical User Interface）、Dispatcher、Coordinator 及模擬引擎（Simulation Engine）四個軟體元件，其中 Coordinator 及模擬引擎共同組成模擬機器（Simulation Machine）。

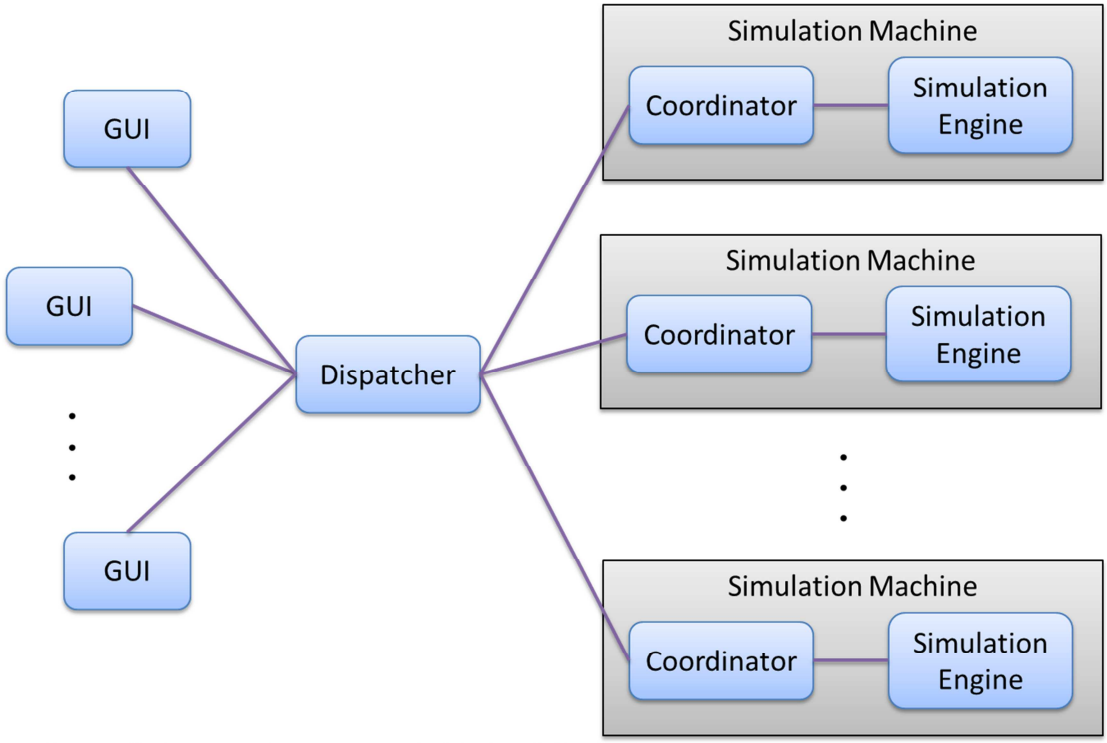

圖 2-1、EstiNet 網路模擬器分散式架構

以下介紹 EstiNet 網路模擬器較重要的元件。

Dispatcher

在 EstiNet 的分散式架構中，Dispatcher 為主要的管控中心，負責分配模擬工作及管理模擬機器。Dispatcher 收到 GUI 送出的模擬工作，從現有模擬機器中挑選空閒的機器執行此模擬工作。Dispatcher 可同時處理多個 GUI 及多個模擬機器。

Coordinator

Coordinator 負責協調與傳遞 GUI、Dispatcher 及模擬引擎之間的訊息。Coordinator 負責將模擬引擎的資訊傳給 Dispatcher 及 GUI，可視為模擬機器對外傳輸資訊的溝通橋樑。

Coordinator 啟動時向 Dispatcher 註冊，表示有一模擬機器可提供模擬服務。Dispatcher 選定一模擬機器進行模擬時，由 Coordinator 接收此資訊並啟動模擬引擎進行模擬，此時模擬機器為忙碌狀態。Coordinator 於模擬進行中亦負責將模擬

引擎的資訊傳給 GUI 讓使用者知道，如模擬進度。模擬完成時 Coordinator 會告知 Dispatcher 模擬工作已結束、此模擬機器回復到空閒狀態，可再次提供服務。

模擬引擎（Simulation Engine）

模擬引擎負責執行模擬工作。模擬引擎使用許多協定模組及修改過的 Linux 核心（kernel），模擬使用者指定的網路拓撲及傳輸並產生記錄檔記錄模擬結果。由於模擬引擎使用核心中無法與其他程序（process）共用的資源，因此在一台機器上只能執行一個模擬引擎。

GUI（Graphical User Interface）

GUI 提供使用者方便的操作介面。使用者用 GUI 繪製網路拓撲、設定網路參數後，GUI 會自動將使用者的設定轉為模擬設定檔，省去使用者自行撰寫相關設定文字檔的不便。這些設定檔會由模擬引擎讀取以進行模擬。

協定模組（Protocol Module）

一個協定模組可視為網路協定堆疊中的一層，由模組開發者依照網路協定的規格實作而成，並在模擬過程中運作各層網路協定的功能。使用者可串連許多協定模組，形成一網路協定堆疊（protocol stack），用來達到不同的實驗、模擬需求。

EstiNet 的分散式架構使它可佈署於多台電腦上，上述 GUI、Dispatcher、Coordinator 皆可分別執行於不同電腦，達到多人使用多個模擬機器的功能。其中 Coordinator 與模擬引擎會共同執行於同一電腦成為模擬機器（Simulation Machine）。在只有一個使用者及一個模擬機器時，也可將所有 EstiNet 的軟體元件執行於同一部電腦進行單機模擬。

2.2 Openstack 介紹

雲端服務分為基礎設施服務（Infrastructure as a Service）、平台即服務（Platform as a Service）、軟體即服務（Software as a Service）三種服務模式。基礎設施服務是直接提供虛擬機器給使用者的雲端服務。平台即服務則提供開發平台給使用者開發程式。軟體即服務則提供應用軟體給使用者使用。

Openstack 為美國 NASA（National Aeronautics and Space Administration）與 Rackspace 公司合作開發的雲端基礎設施服務（Infrastructure as a Service）之雲端運算軟體。

Openstack 中，一台虛擬機器（Virtual Machine）由硬碟映像檔（disk image）及 flavor 組成。硬碟映像檔是虛擬機器的硬碟，以檔案的形式儲存於實體機器中。Flavor 是設定一台虛擬機器有多少運算資源，如多少 CPU、記憶體。

雲端系統架設者可透過指令或 Openstack API（Application Programming Interface）使用 Openstack。Openstack API 是一個 RESTful Web 服務（RESTful Web Service）。在 RESTful Web 服務中，使用者透過傳送 HTTP 要求（HTTP request）存取及使用資源，接收 HTTP 回應（HTTP response）得知存取、執行等結果。

使用者須通過 Openstack 的認證才可使用 Openstack 的服務。要透過 API 使用 Openstack 的服務時，使用者須先用帳號密碼向 Openstack 取得一個認證 token，通常為一亂數字串。之後傳送 API 要求時附上認證 token，Openstack 就會將該要求視為合法要求。認證 token 有使用時間的限制，如果超過使用時間 Openstack 不會再認定該要求是合法的，此時使用者須再次傳送帳號及密碼取得新的認證 token。使用 API 操作時會有許多 HTTP 要求及回應，Openstack 以此 token 認證方式避免使用者的帳號資訊夾帶在各種 HTTP 要求中，降低使用者帳號資訊外洩的可能。

Openstack 以資料庫儲存系統的相關資訊及狀態，例如虛擬機器、Fixed IP Address 及 Floating IP Address 等資訊，Fixed IP Address 及 Floating IP Address 將

於章節 2.2.2 說明。Openstack 在做各種操作時，會不斷更新其資料庫中的資料，資料庫也是各種查詢的資料來源。

2.2.1 Openstack 軟體元件介紹

Openstack 為一龐大系統，由許多模組各自負責不同服務，模組又由許多元件構成。以下僅介紹與我們網路模擬雲端系統較相關的模組及其重要元件。

Nova

Nova 是 Openstack 最重要的模組。它由多個元件組成，主要負責管理計算資源，如開啟、關閉、分配虛擬機器等操作。在我們使用的 Openstack 版本中，虛擬機器的網路相關設置也是由 Nova 負責。以下介紹 Nova 中較重要的元件。

Nova-Compute

Nova-Compute 為一程序（process）。它會接收開啟或關閉虛擬機器的要求，使用下層虛擬機器 hypervisor 的 API 來開啟或關閉虛擬機器，最後將資訊寫入資料庫。

Nova-Network

Nova-Network 是處理虛擬機器網路相關設定的程序。其接收網路相關要求，執行對應的操作，如設置網路介面（network interface）、修改 iptables 規則等。Openstack 網路架構將在章節 2.2.2 說明。

Nova-Api

Nova-Api 負責接收及回應使用者的 Openstack API 要求。

Nova-Scheduler

Openstack 要開啟虛擬機器時，會由 Nova-Scheduler 以排程演算法挑選用於開啟虛擬機器的實體機器。雲端系統架設者可設置不同的排程演算法，讓虛擬機器可以不同方式分散在雲端眾多實體機器中。

Glance

Glance 是負責管理虛擬機器映像檔（Virtual Machine Image）的模組，例如儲存、取出映像檔。虛擬機器映像檔是開啟虛擬機器時所需要的硬碟映像檔（disk image）範本，虛擬機器開啟後硬碟最初的內容會是此範本的內容。

上述模組、元件可依照雲端系統架設者的需求執行在不同機器上。

2.2.2 Openstack 網路架構

以 Openstack 建立的雲端系統中含有多台實體機器與多台虛擬機器，這些機器透過內部網路連接。內部網路主要由 Openstack 的 Nova 模組設定及管理。Openstack 對於內部網路有三種網路組態可供選擇，分別為 Flat、Flat DHCP、VLAN。無論使用哪種組態，實體機器都需要兩張網路介面卡（network interface）。一張負責機器本身的對外連線，一張負責與虛擬機器溝通。負責與虛擬機器溝通的網路介面卡稱為內部網路介面卡（internal network interface）。

Openstack 有 single-host、multi-host 兩種網路管理模式。兩種模式中虛擬機器與外界網路的連線、網路流量皆會通過 Nova-Network。在 single-host 模式下，整個網路中只有一個 Nova-Network。在 multi-host 模式中，每台運行 Nova-Compute 的實體機器皆會執行 Nova-Network。

我們使用 Flat DCHP 網路組態及 single-host 管理模式。Flat DHCP 中，每台主機需建立一虛擬橋接器（bridge），此為 Linux 系統虛擬出來的網路介面。一台主機上所有虛擬機器及此主機的內部網路介面卡會連接到此橋接器，本論文於章節 3.2 說明實際建置的狀況。另有一稱為 dnsmasq 的小型 DHCP server 監聽在此虛擬橋接器上，負責分配 IP Address 給虛擬機器。

Openstack 將虛擬機器的 IP Address 區分為 Floating IP Address 與 Fixed IP Address。Fixed IP Address 是虛擬機器建立、開啟後得到的 IP Address，在虛擬機器被關閉、刪除前都擁有此 IP Address。Floating IP Address 則可以在任何時間動態關聯（associate）給虛擬機器或從虛擬機器上卸離（disassociate）。

在應用上，提供雲端服務的廠商會將 Public IP Address 作為資源販賣或租借給使用者使用。使用者可能有多台虛擬機器，但只有一個 Public IP Address，如果能將 Public IP Address 動態的配置給不同虛擬機器，對使用者會有較大的彈性。透過 Openstack 的 Fixed IP Address 及 Floating IP Address 機制，廠商可以把 Fixed IP Address 設定為 Private IP Address，Floating IP Address 設定為 Public IP Address。如此，在雲端內部虛擬機器的網路組態可以用 Private IP Address 管理，而 Public IP Address 就能作為資源賣給使用者。使用者可以將他購買的 Public IP Address 任意關聯到不同的虛擬機器上，IP Address 不會受限於一台虛擬機器。

在我們的系統中，Fixed IP Address 跟 Floating IP Address 都是 Private IP Address。Fixed IP Address 由 Openstack 分配給虛擬機器，是 192.168.4.0 網段的 IP Address，用於虛擬機器間的溝通。我們的網路架構除了虛擬機器的網段之外，另有自行設置的 10.10.10.0 內部網段，我們其他的實體機器跟服務會架設在此網段。虛擬機器存取其他實體機器的服務時，它的 Fixed IP Address 會被轉換成對應的 Floating IP Address。因此，我們將 Floating IP Address 設為 10.10.10.0 的網段，讓虛擬機器能存取其他實體機器的服務，如 NFS。

Openstack 透過 iptables、route 等工具設定各台實體機器的網路組態，以管理雲端的內部網路。例如，Floating IP Address 的關聯是在相關實體機器上設置 iptables 的 NAT 規則，針對虛擬機器的封包做 Fixed IP Address 及 Floating IP Address 轉換，此部分將於章節 3.2 說明實際建置的狀況。

第3章 系統架構

我們在 EstiNet 的分散式架構基礎上修改及增加功能，配合 Openstack 所提供的的基礎設施服務，設計、實作出 EstiNet 網路模擬雲端系統。在我們的系統架構中，使用虛擬機器作為運算單元、執行 EstiNet 網路模擬器以執行模擬工作，並利用 Openstack 管理虛擬機器資源。

此網路模擬雲端系統中，模擬工作分為模擬工作（Job）及模擬子工作（Subjob）。一個模擬工作可能包含一個或多個模擬子工作，各子工作可分別獨立進行模擬，此部分將於章節 4.2 進一步說明。

在本章中，我們先介紹整體系統的主要架構。先介紹軟體階層的架構，接著說明實體機器與網路架構，最後說明各軟體元件分別運行於哪些實體機器中。

3.1 系統軟體元件架構

首先我們以圖 3-1 說明 EstiNet 網路模擬雲端系統的軟體元件架構圖。其中的大框表示雲端系統部分，區分雲端系統的內外。我們將 GUI 置於雲端之外，其餘元件則位於雲端內部，系統內外以 Forwarder 作為溝通橋梁。雲端內部以 Manager 為主要控制中心。

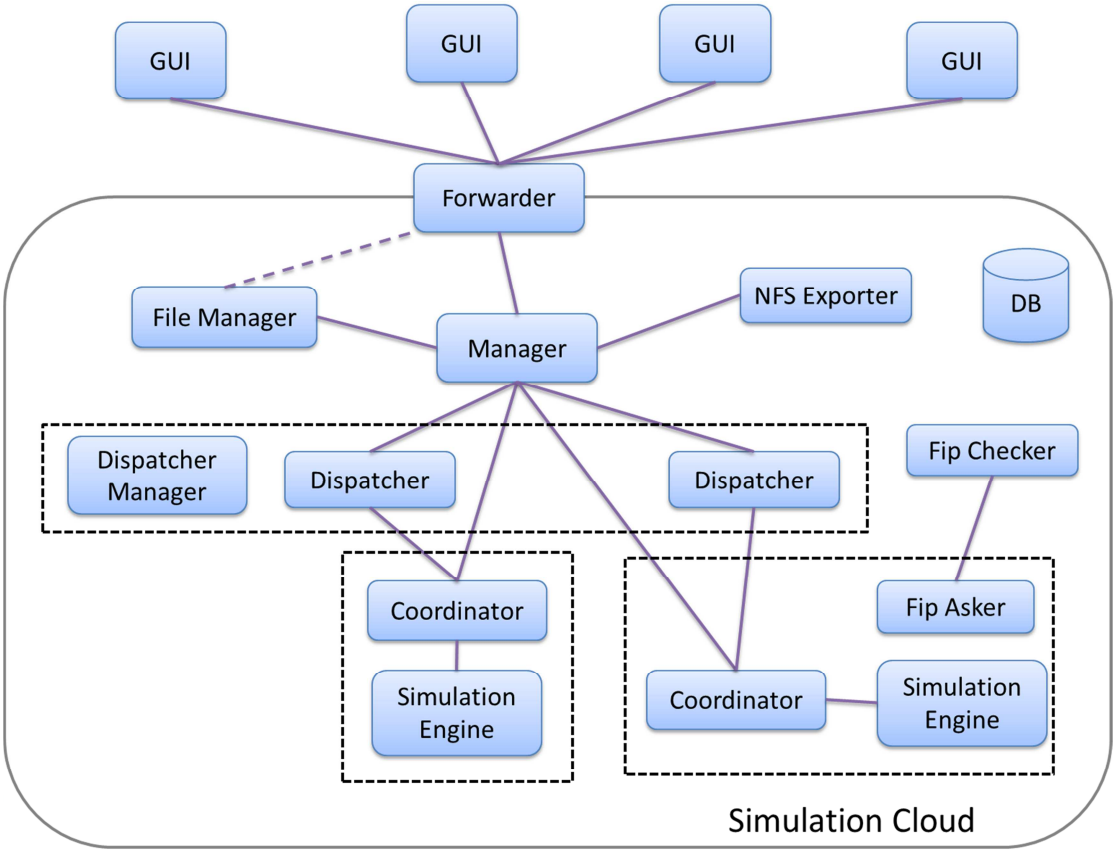

圖 3-1、網路模擬雲端系統軟體元件架構

圖 3-1 中虛線方框及其中元件表示這些元件執行在同一虛擬機器中，此部分將於章節 3.3 進一步說明。線段代表各元件之間的連線，實線表示元件啟動或連上雲端系統時便會建立的連線，虛線則是依據要求建立。以下先介紹各元件的主要功能，之後再說明軟體架構。

GUI

GUI 為 EstiNet 網路模擬器圖形操作介面。我們讓使用者以 GUI 直接使用雲端模擬服務，作為 EstiNet 網路模擬雲端服務的客戶端。

Forwarder

雲端內部及外部的溝通、轉送橋樑，作為雲端系統的主要出入口。

Manager

雲端系統的控制中樞，主要負責接收與分配模擬工作、開啟及關閉虛擬機器。Manager 透過 Openstack API 開啟及關閉虛擬機器、關聯（associate）及卸離（disassociate）Floating IP Address。

File Manager

File Manager 負責處理與檔案及資料庫有關的操作，例如接收模擬設定檔、傳送模擬結果檔、查詢資料庫。

Dispatcher

Dispatcher 負責產生模擬子工作、分配子工作、管理模擬資源、處理與模擬工作有關的資訊並更新至雲端系統資料庫。功能與 EstiNet 原始的 Dispatcher 雷同，但不再與 GUI 直接連線，模擬要求由 Manager 傳送給 Dispatcher 。

Dispatcher Manager

在一台執行 Dispatcher 的虛擬機器中會執行多個 Dispatcher，Dispatcher Manager 負責讓虛擬機器中有固定個數的 Dispatcher。

Coordinator

Coordinator 在雲端系統中僅負責執行模擬引擎及扮演模擬引擎及 Dispatcher 間的溝通橋樑，不再如原有分散式架構與 GUI 直接連線，所有要傳輸給 GUI、讓使用者知道的資訊會交由 Dispatcher 處理。

模擬引擎（Simulation Engine）

模擬引擎功能與 EstiNet 原始版本相同，負責執行模擬。

資料庫（Database）

資料庫用於儲存網路模擬雲端系統所使用的資訊、使用者認證所需的使用者資訊以及模擬工作相關資訊，如工作狀態、進度等。由於此系統為一雲端系統，如遭遇各硬體、軟體之錯誤，系統應能在重新啟動後恢復先前的運作狀態，因此我們將軟體元件的狀態存於資料庫以供發生錯誤時回復，這些資訊如虛擬機器分

配及使用的狀態。由於 Openstack 平台中也有資料庫，後面章節如無特別描述，資料庫皆指我們雲端系統本身的資料庫。

NFS Exporter

NFS Exporter 負責動態分享 NFS（Network File System）的目錄給虛擬機器掛載（mount），因此 NFS Exporter 必須執行在架設 NFS 的主機中。此元件的功能及運作將於章節 4.4 進一步說明。

Fip Checker 及 Fip Asker

用以確認虛擬機器是否已獲得 Floating IP Address，詳細說明於章節 4.2.3。

由圖 3-1 可以看到，我們將 GUI 分離至雲端系統外，這是由於如果將 GUI 放在雲端系統內部，讓使用者以 VNC（Virtual Network Computing）程式遠端連線至雲端內的 GUI 操作及進行模擬，在操作上會有難以忍受的延遲感以及大量的網路傳輸。

整個雲端系統只能透過 Forwarder 連外。Forwarder 是我們自行撰寫的轉送程式，因此雲端內部元件無法直接與網際網路連線，網際網路中的機器也無法直接與雲端內部元件連線。我們以此方式減少雲端內部與外界連線，作為第一道安全性防護，不僅保護雲端，也避免惡意使用者使用虛擬機器攻擊外部網路。

在軟體架構上，我們以同步（synchronize）的方式處理各種要求（request）。軟體元件可能需要執行多個步驟才能完成一個要求，元件需完成一個要求的所有步驟才能再處理下一個要求。

例如，GUI 傳送模擬工作至雲端時，會由 Manager 及 File Manager 分別接收模擬工作要求及模擬設定檔。這三個元件分別需完成幾個步驟，才算完成傳送、接收模擬工作的要求：GUI 需完成對 Manager 傳送模擬工作要求以及對 File Manager 完成檔案傳輸等步驟。Manager 需完成接收模擬工作要求、儲存相關資訊等步驟。File Manager 則需要完成接收模擬設定檔、通知 Manager 等步驟。

當一個要求由多個元件共同完成時，它們彼此需要傳遞控制訊息及資料，我們用 socket 連線達到軟體元件間的溝通，即 IPC（Inter-Process Communication）。

如圖 3-1 所示，雲端系統中 Manager 與 File Manager、Forwarder、NFS Exporter、Dispatcher、Coordinator 間有控制連線，此連線用於傳送各種 IPC 命令及回應。Manager、File Manager、Forwarder 及 NFS Exporter 是此雲端系統運作時最初需啟動的程序（process），這些元件啟動時便會建立控制連線。Dispatcher 與 Coordinator 則是在要提供模擬服務時才會執行，這兩個程序啟動時會主動連向 Manager 以建立控制連線。透過與 Dispatcher 及 Coordinator 的連線，Manager 可知道系統中有多少 Dispatcher 及 Coordinator 正在運作並可加以管理。

Forwarder 與 Manager 間除了控制連線外仍有多條用於傳輸 GUI 資料的連線。而 Forwarder 與 File Manager 間的連線在 GUI 要與 File Manager 連線時才會建立，同樣用於 GUI 資料的傳輸。此部分於章節 4.2.1 進一步說明。

Dispatcher 與 Coordinator 間的連線在 EstiNet 網路模擬器原有架構中即存在，用於傳輸控制命令。在雲端系統中我們保留此連線、加入雲端系統需要的 IPC，並修改建立連線的部分，此修改於章節 4.2.3 說明。

Coordinator 與模擬引擎間的連線在 EstiNet 網路模擬器原有架構中即存在，同樣用於傳輸控制命令。在雲端系統中我們保留此連線及其用途，並加入雲端系統所需的 IPC。

Fip Checker 也是在雲端系統最初運作時會啟動的程序。Fip Checker 與 Fip Asker 間的連線是在 Fip Asker 啟動時主動連向 Fip Checker，此部分於章節 4.2.3 進一步說明。

上述 socket 連線中，Coordinator 與模擬引擎間使用 Unix domain socket，其餘皆為 TCP socket。

上述軟體元件依據其負責項目及功能的不同，實作上有不同細節。以程式架構來看，各軟體元件主要執行流程為：

1. 等待指令。

2. 從連線收到指令。

3. 依據指令執行相應步驟，完成後回到 1。

若執行的步驟可能造成程序被阻擋（block），則可能 fork 出子程序執行該步驟，避免整個程序被一個要求阻擋（block）而無法進行其他動作。

3.2 實體機器與網路架構

圖 3-2 為網路模擬雲端系統的實體架構、線路圖。其中 Front-end、Worker、Controller、Node*皆為實體機器。NFS 及資料庫則架設於在 Worker 中。如 Openstack 介紹中所述，Openstack 內部網路中的實體機器皆需兩張網路介面卡，一負責對外連線，一為負責與虛擬機器溝通、處理虛擬機器的網路傳輸。因此 Controller 及所有 Node 皆有兩張網路介面卡，分別連接到兩台 Switch。以下說明各台機器的用途。

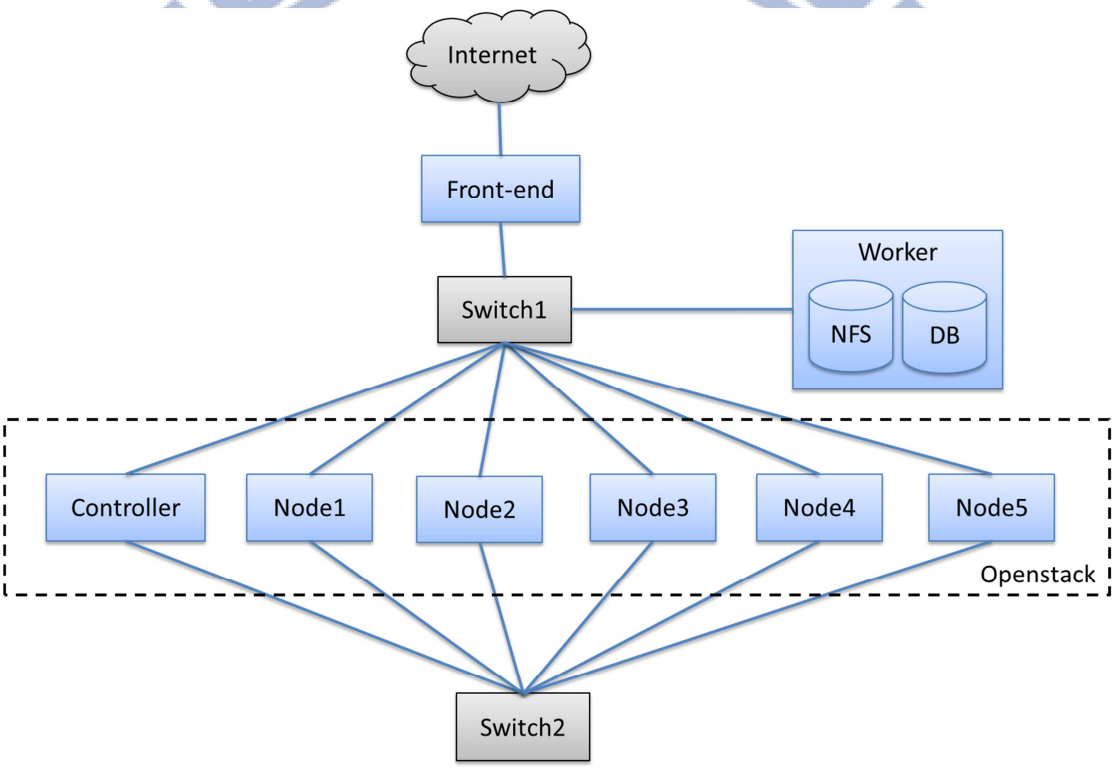

圖 3-2、實體機器架構圖

Front-end

網路模擬雲端系統的連外閘道，所有連線皆須經過此機器。在我們的實際建置上目前僅有一台 Front-end，但在架構及軟體設計上若配合 DNS 的伺服器負載平衡（Load Balancing），則可有多台 Front-end。本論文中以一台 Front-end 說明。

Worker

網路模擬雲端系統的控制中心，Manager、File Manager 等主要控制元件的執行處。資料庫及 NFS 架設於此。

Controller

Openstack 的控制中心，Nova-Network、Nova-Scheduler、Nova-Api、Glance 等服務皆運行於此。Controller 僅負責 Openstack 的控制與管理，不會執行 Nova-Compute 提供運算資源。

Node*

提供運算資源，用以開啟虛擬機器。Nova-Compute 服務執行於此。

此雲端系統中的網路是一封閉的內部網路，連接各虛擬機器及實體機器。接下來我們說明此雲端系統的內部網路架構。圖 3-3 為雲端系統內部網路邏輯架構圖。此雲端系統的內部網路分為兩層，外層是我們自行設置的網路（圖 3-3 上半部），內層是由 Openstack 建立之虛擬機器內部網路（圖 3-3 下半部），內外層分屬不同網段，而這兩部分以 Controller 連接。虛擬機器與外層網路中的機器進行傳輸或經由 Front-end 連向網際網路時，其網路傳輸皆會通過 Controller。

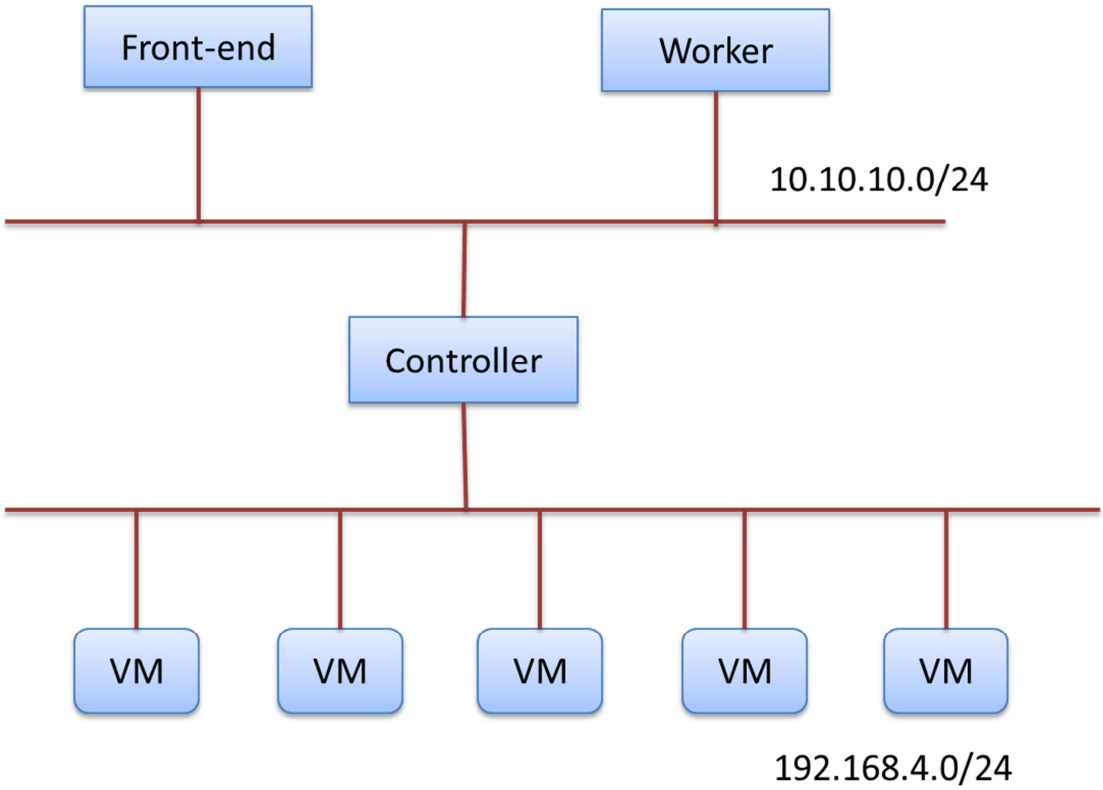

圖 3-3、雲端系統內部網路架構

在雲端系統中，虛擬機器會開啟於多台實體機器，即我們系統中的 Node。虛擬機器的網路傳輸會透過實體機器的網路介面卡往外傳送，因此虛擬機器與實體機器須有相關網路組態設置，以使虛擬機器的流量可以先傳送到實體機器，再由實體機器往外傳送。此組態設置如圖 3-4 所示，一個 Node 中有多台虛擬機器，每台虛擬機器的網路介面卡 eth0 及 Node 的網路介面卡 eth1 會橋接至 Node 虛擬出來的橋接器（bridge）br100 上。因此虛擬機器的網路傳輸會經由它的 eth0 傳到 br100，再由 Node1 的 eth1 往外傳輸，Node 的 eth1 即為內部網路介面卡。

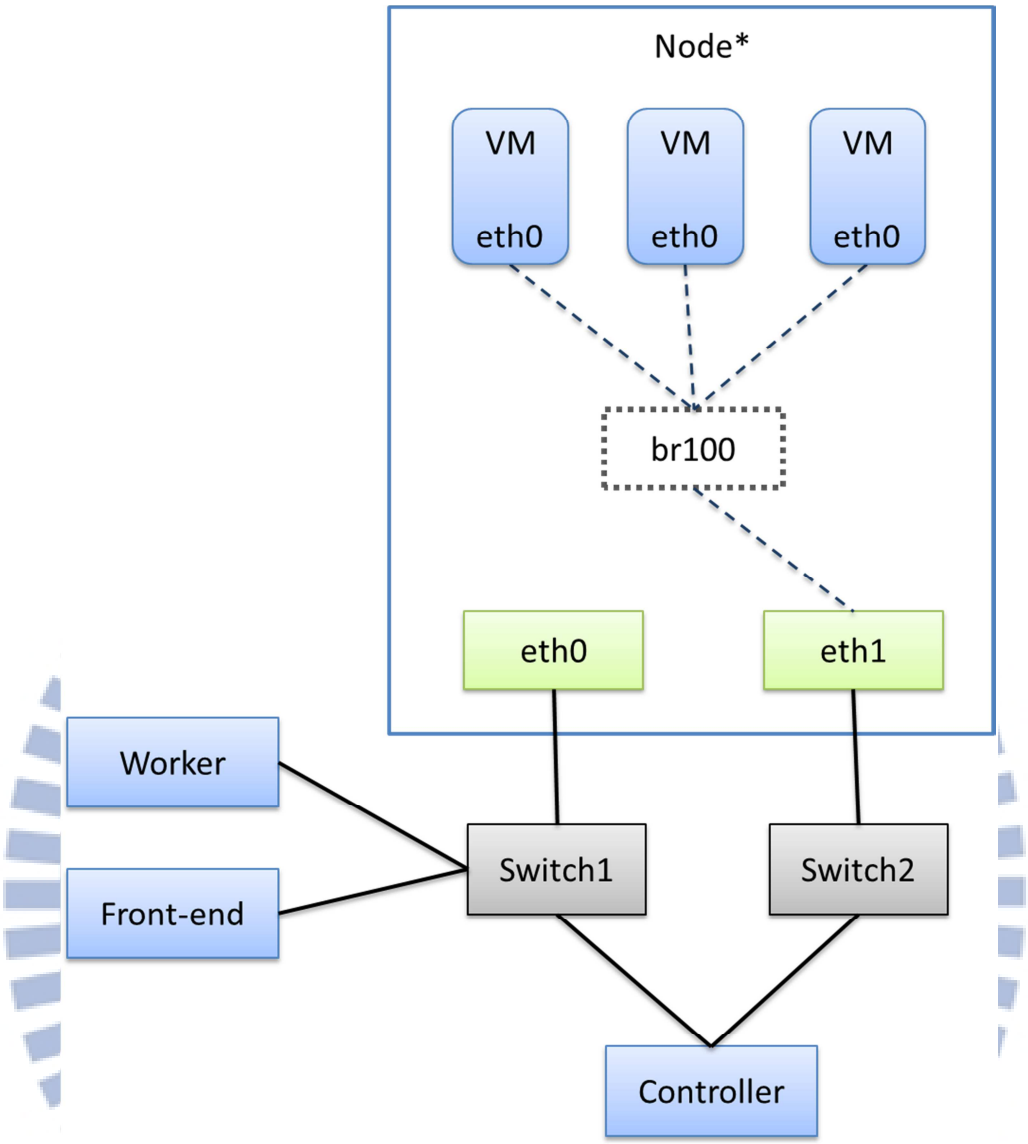

圖 3-4、Node 與虛擬機器的網路橋接

在背景介紹中提到，Openstack 對虛擬機器的內部網路有多種組態及管理模式，我們使用 Flat DCHP 組態及 single-host 管理模式作為內部網路組態。我們將 Nova-Network 執行在 Controller 上，因此所有虛擬機器對外的網路流量會經過 Controller。我們以圖 3-5 說明虛擬機器對外傳輸的封包流向。封包從虛擬機器的網卡 eth0 傳至 Node 的虛擬橋接器 br100。接著流向 Node 的網卡 eth1，再傳至 Switch2。Controller 的內部網路介面卡收到後，經過 Controller 的處理再將封包往外傳至 Switch1。最後傳至 Worker 或者透過 Front-end 傳送到網際網路。

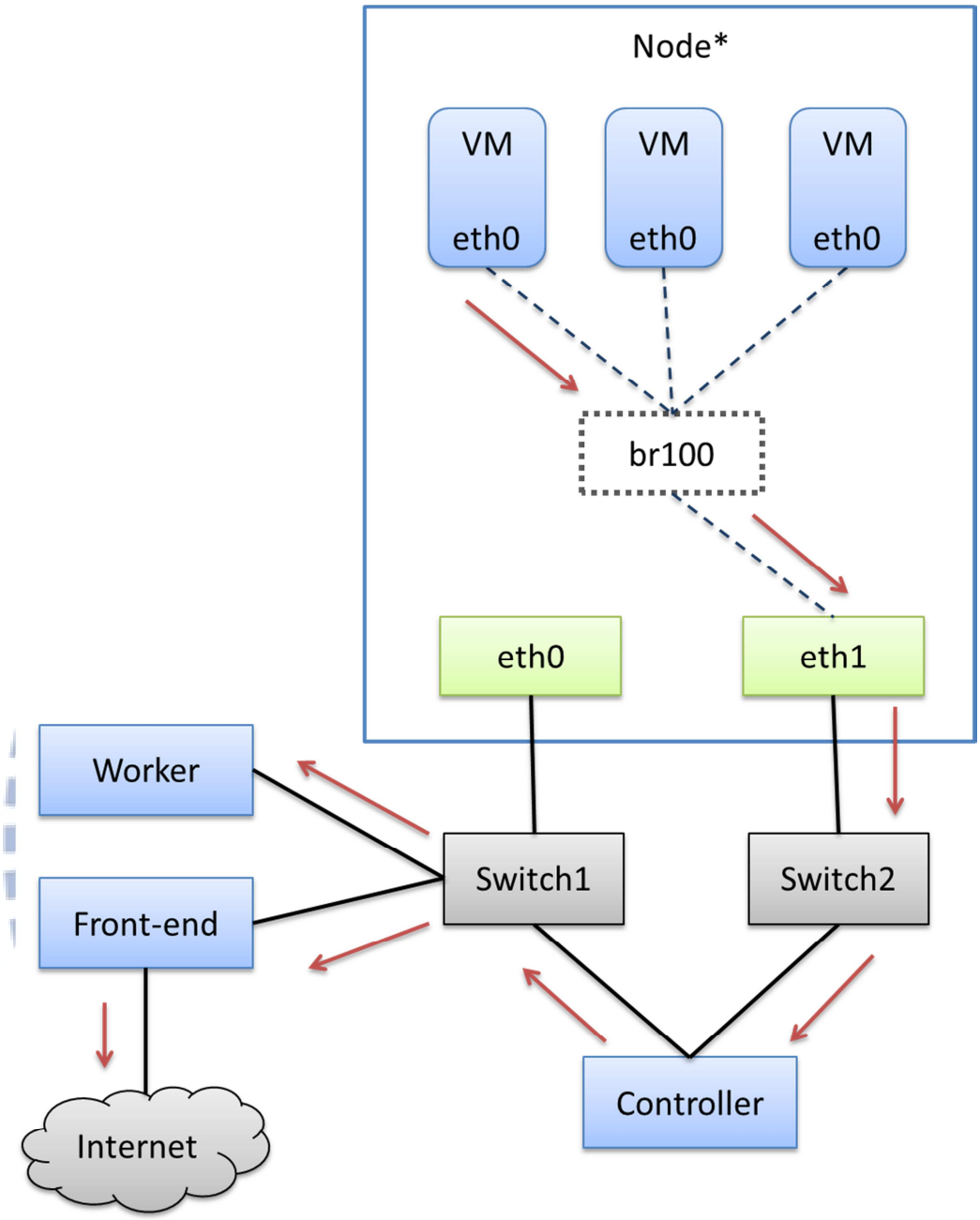

圖 3-5、虛擬機器對外傳輸的封包流向

Openstack 網路架構中所提到的 Floating IP Address 關聯機制在我們的網路架構中，是在 Controller 上以類似 NAT（Network Address Translation）的機制達成。虛擬機器的封包往外傳、流經 Controller 時，Controller 會將封包的來源 IP Address 從虛擬機器的 Fixed IP Address 改為 Floating IP Address。當有封包要傳給某台虛擬機器，該封包流經 Controller 時，Controller 會將目的 IP Address 從虛擬機器的 Floating IP Address 改為 Fixed IP Address，讓虛擬機器可正確接收。因此虛擬機

器本身無法直接得知自己的 Floating IP Address，只能知道 Fixed IP Address。而對於非內層網路的實體機器來說，如 Worker、Front-end，它們得知的虛擬機器 IP Address 會是 Floating IP Address 而非 Fixed IP Address。

在我們的設置中，如圖 3-3，192.168.4.0 網段是 Fixed IP Address，而 10.10.10.0 網段是 Floating IP Address。虛擬機器存取 Worker 上的服務時，Controller 會將虛擬機器的 Fixed IP Address 轉換為這台虛擬機器對應的 Floating IP Address。這個 Fixed IP Address 及 Floating IP Address 轉換機制影響到我們部分元件的設計與實作，此將於章節 4.2.3 說明。

3.3 系統軟體元件與實體機器之關係

綜合前面兩個小節所說明的各軟體、硬體之功能，圖 3-6 表示出我們的軟體元件分別執行於哪些實體及虛擬機器。

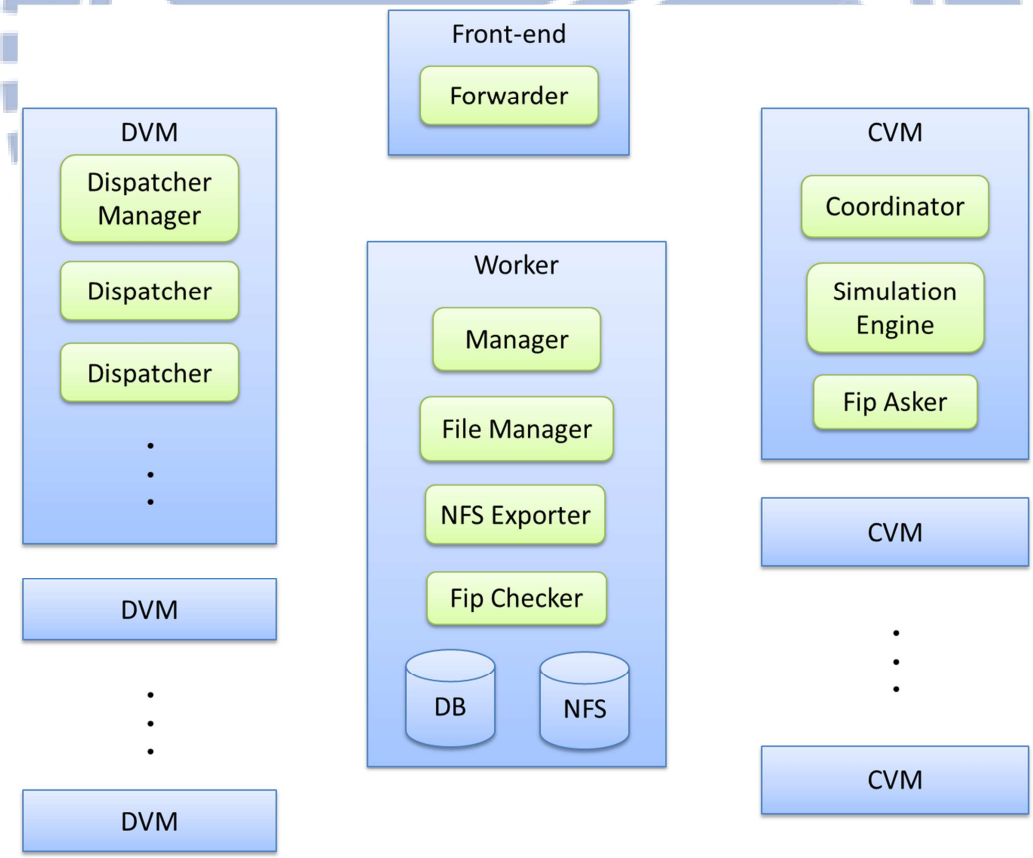

圖 3-6、軟體元件與實體機器之關係

實體機器部分，Forwarder 執行於 Front-end，擔任整個雲端系統對外連線的閘口。Manager、File Manager、NFS Exporter、Fip Checker 執行於 Worker，資料庫及 NFS 架設於 Worker。由於 NFS 是架設在 Worker 上，File Manager 是直接在 Worker 的硬碟上進行檔案操作。

我們將虛擬機器分為兩種：Dispatcher VM（DVM）及 Coordinator VM（CVM）。DVM 負責執行 Dispatcher。由於 Dispatcher 使用的系統資源較少，一台 DVM 可執行許多 Dispatcher，可讓虛擬機器的運算資源，如 CPU 及記憶體，達到較有效的利用。為了讓 DVM 中 Dispatcher 的數量可以維持在一定數目，我們在 DVM 中執行一 Dispatcher Manager 負責此工作。CVM 中運行 Coordinator 及模擬引擎，是主要進行計算的單位。CVM 中另有 Fip Asker，用來確認 CVM 已取得 Floating IP Address，此部分將於章節 4.2.3 說明。

在虛擬機器運算資源的配置上，如記憶體大小，可依照不同的需求分別設定，以達到運算資源的有效利用。一般來說執行模擬工作的 CVM 會需要較多運算資源，而僅執行 Dispatcher 的 DVM 需要的運算資源較少，我們可以調整不同的運算資源設置以符合需求及達到資源的有效利用。

第4章 設計與實作

4.1 設計目標與系統設計

在學術研究、網路協定的設計與研發上，研究者通常會以不同參數進行多次實驗，觀察在不同環境、參數下網路協定的運作狀況，進而修正、改良。在 EstiNet 網路模擬器原有的使用方式中，研究者需手動指定各種模擬參數來進行多次實驗。

我們網路模擬雲端系統希望讓使用者以簡單的操作在較短的時間完成多個模擬工作，省去使用者修改各種網路參數的麻煩，同時縮短使用者執行多項模擬的總時間。另外，由於 EstiNet 網路模擬器可讓使用者自行開發網路協定模組，因此在雲端系統中我們同樣希望使用者可使用自行開發的協定模組及應用程式來執行模擬。

為了達到上述設計目標，我們設計並實作「併行模擬」、「使用者自訂模擬引擎及應用程式」、「儲存空間管理」、「模擬引擎錯誤處理」、「認證機制」。這五大部分的目標如下所述。

我們以「併行模擬」讓使用者以簡單的方式指定網路參數變化，雲端系統自動產生多個類似但些許參數不同的模擬子工作，並自動用多台虛擬機器同時進行這些模擬子工作。我們以「使用者自訂模擬引擎及應用程式」提供使用者使用其自行開發的網路協定模組及應用程式。

「併行模擬」執行模擬工作會產生模擬結果檔案，使用者自行開發的應用程式及模擬引擎需有儲存空間存放。我們不希望使用者的結果檔案毫無限制的儲存於雲端系統中，因此雲端系統中讓每位使用者擁有自己的儲存空間存放檔案並限制使用者的空間配額。此設計延伸出空間配額、檔案權限管控等議題，我們在「儲存空間管理」說明此部分。

使用者可使用自行開發的網路協定模組是此系統的功能之一，但如果使用者程式撰寫不當可能造成雲端系統中的模擬引擎發生錯誤。模擬引擎在雲端系統的虛擬機器中發生錯誤時，使用者無法如一般單機上可直接介入、取得錯誤資訊、重新啟動，因此雲端系統須有相關機制處理此問題，即為「模擬引擎錯誤處理」。

最後，EstiNet 網路模擬器為一商業軟體，使用者使用某些功能進行模擬前須通過一認證機制，確認使用者有購買該項功能。由於雲端系統架構的限制，無法以原有機制進行認證，因此我們須另外處理雲端系統中與認證有關的問題，此以「認證機制」達成。

4.2 併行模擬

併行模擬是此網路模擬雲端系統中的主要功能。此功能讓使用者可指定網路參數變化，接著雲端系統自動產生多個類似但些許參數不同的模擬子工作，並自動以用多台虛擬機器同時進行這些模擬子工作。使用者僅須設定並執行一個模擬工作，便可得到在單機模式中需要多次設定、執行模擬的數據及結果，將節省使用者實驗所需的龐大時間。

在此目標下，我們設計每個模擬工作會有一或多個模擬子工作，每個模擬子工作有相同的網路拓撲，但分別有不同的網路參數值，如網路頻寬（bandwidth）。每個模擬子工作實際上為一可獨立進行的模擬，因此可由一虛擬機器執行模擬。當有多個模擬子工作時便可以多台虛擬機器同時進行模擬，如此可減少整個模擬工作所需要的時間。

併行模擬的使用上，使用者先以 GUI 繪製網路拓撲，並設定相關網路參數及欲使用的應用程式，此步驟為使用 EstiNet 網路模擬器進行網路模擬的基本步驟。完成相關設定後，使用者可使用雲端服務執行模擬工作。一個模擬工作在網路模擬雲端系統的執行流程為：

1. 使用者用 GUI 登入雲端服務。

2. 使用者設置相關參數並傳送模擬工作至雲端。

3. 雲端系統收到模擬工作後，分配虛擬機器給此模擬工作。

4. 雲端系統依照設定產生模擬子工作。

5. 雲端系統開始以多台虛擬機器同時執行多個子工作的模擬運算。

6. 模擬過程中使用者可透過 GUI 取得模擬工作及使用資源的相關資訊，例如工作名稱、模擬進度、使用的模擬機器數量。

7. 所有模擬子工作完成後，雲端系統釋放此工作所佔用的運算資源，使用者可透過 GUI 取得模擬結果。

8. 使用者也可於模擬進行中、結束後，對模擬工作進行相關操作。

接下來各小節將說明上述步驟的實作方式。

4.2.1 連線及登入

使用者用 GUI 連接網路模擬雲端系統。在系統架構中提到，Forwarder 為雲端系統對外的出入閘口，GUI 與雲端內部元件的連線需透過 Forwarder 轉送。以下說明 GUI 透過 Forwarder 與 Manager、File Manager 建立連線及轉送機制。

要使用雲端服務，使用者需先登入、通過認證，而認證是透過 Manager 進行，因此 GUI 最初需先與 Manager 建立連線。如圖 4-1 所示，GUI 與 Manager 透過 Forwarder 建立連線的流程：

1. GUI 與 Forwarder 建立連線，傳送與 Manager 建立連線的要求。

2. Forwarder 為此連線產生一連線識別資料，此識別資料用於後面配對連線步驟。因為 Forwarder 可能同時有許多連線需進行配對，需能識別每條連線。我們以時間及一計數值作為識別資料。

3. Forwarder 透過與 Manager 間的控制用連線傳送連線要求，並將剛產生的識別資料傳給 Manager。

4. Manager 主動連向 Forwarder、建立一條新的連線，此連線於橋接完成後將

視為與 GUI 的連線。

5. Forwarder fork 出子程序 Connector，子程序 Connector 會有父程序 Forwarder 所有的連線。Connector 為實際上進行轉送的程序。

6. Connector 依據兩邊的識別資料進行配對，關閉其他無關的連線，僅留下兩條識別資料相同的連線。Forwarder 則關閉兩條識別資料相同的連線。

7. Connector 通知 GUI 及 Manager 橋接已完成。此後 GUI 將視原與 Forwarder 的連線為與 Manager 的連線，Manager 將視第 4 步中與 Forwarder 新建立的連線為與 GUI 的連線。

8. 完成橋接後，Connector 僅作轉送，將 GUI 傳送來的資料送往 Manager、將 Manager 傳送來的資料送往 GUI。

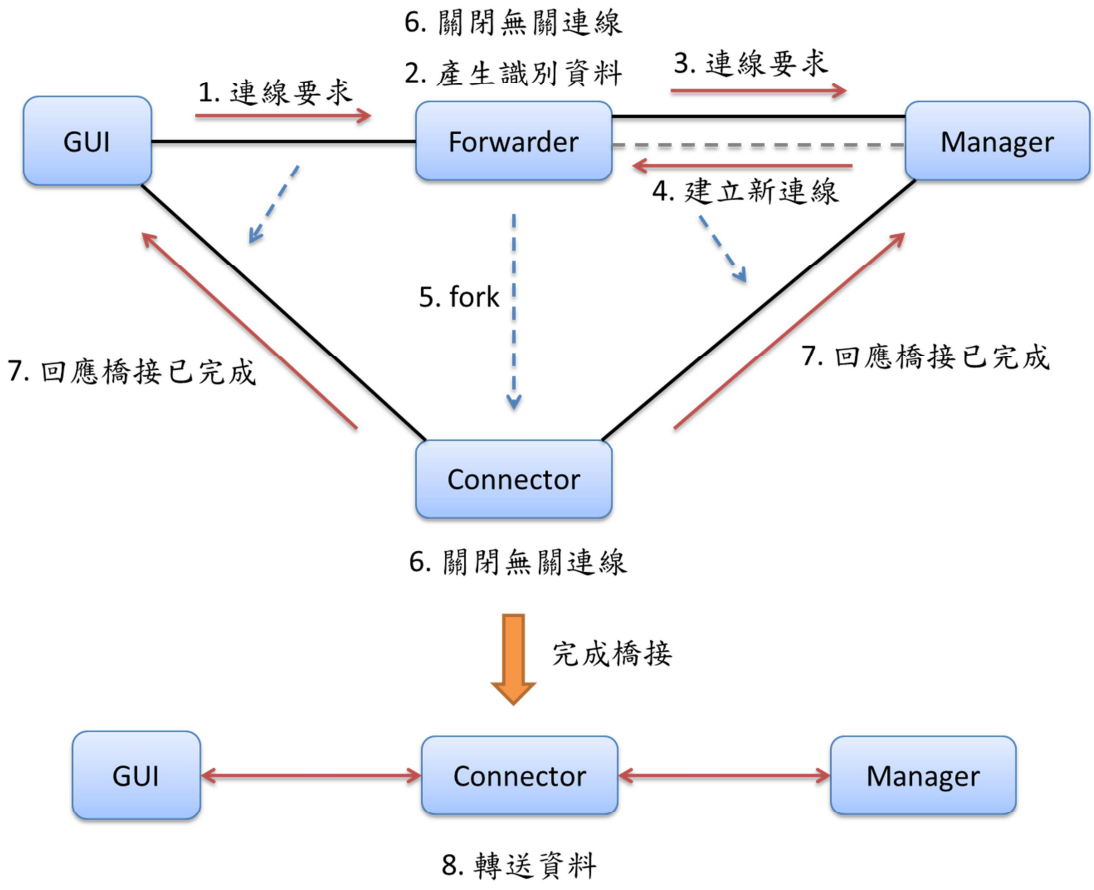

圖 4-1、GUI 與 Manager 透過 Forwarder 建立連線

GUI 需要傳輸檔案或查詢資料時，會與 File Manager 建立連線以進行這些操作。因為 File Manager 不像 Manager 與 Forwarder 間有一條控制連線，GUI 與 File Manager 的連線要求需透過 Manager 傳給 File Manager。因此，GUI 要與 File Manager 建立連線前必須先與 Manager 連線並通過登入認證。如圖 4-2 所示，GUI 與 File Manager 透過 Forwarder 建立連線的流程：

1. GUI 與 Forwarder 建立連線，傳送與 File Manager 建立連線的要求。

2. Forwarder 為此連線產生一連線識別資料，並將此識別資料回傳給 GUI。

3. GUI 傳送與 File Manager 建立連線的要求及連線識別資料給 Manager。

4. Manager 將連線識別資料傳送給 File Manager，要求 File Manager 向 Forwarder 建立連線。

5. File Manager 回應 Manager，表示有收到此要求。

6. File Manager 向 Forwarder 建立一連線並傳送連線識別資料。此連線識別資料是第 2 步時 Forwarder 為 GUI 的連線產生的，我們透過將識別資料經由以上步驟傳送至 File Manager 再傳給 Forwarder，用以配對 GUI 與 File Manager 的連線。

7. Forwarder fork 出子程序 Connector。Connector 為實際上進行轉送的程序。

8. Connector 依據識別資料進行配對，關閉其他無關的連線，僅留下兩條識別資料相同的連線。Forwarder 則關閉兩條識別資料相同的連線。

9. Connector 通知 GUI 及 File Manager 橋接已完成。此後 GUI 將視原與 Forwarder 的連線為與 File Manager 的連線，File Manager 將視第 6 步中與 Forwarder 新建立的連線為與 GUI 的連線。

10. 完成橋接後，Connector 僅作轉送，將 GUI 傳送來的資料送往 File Manager、將 File Manager 傳送來的資料送往 GUI。

圖 4-2 中 GUI 與 Manager 的連線實際上有 Connector 位於中間轉送，在此簡化圖示表示。

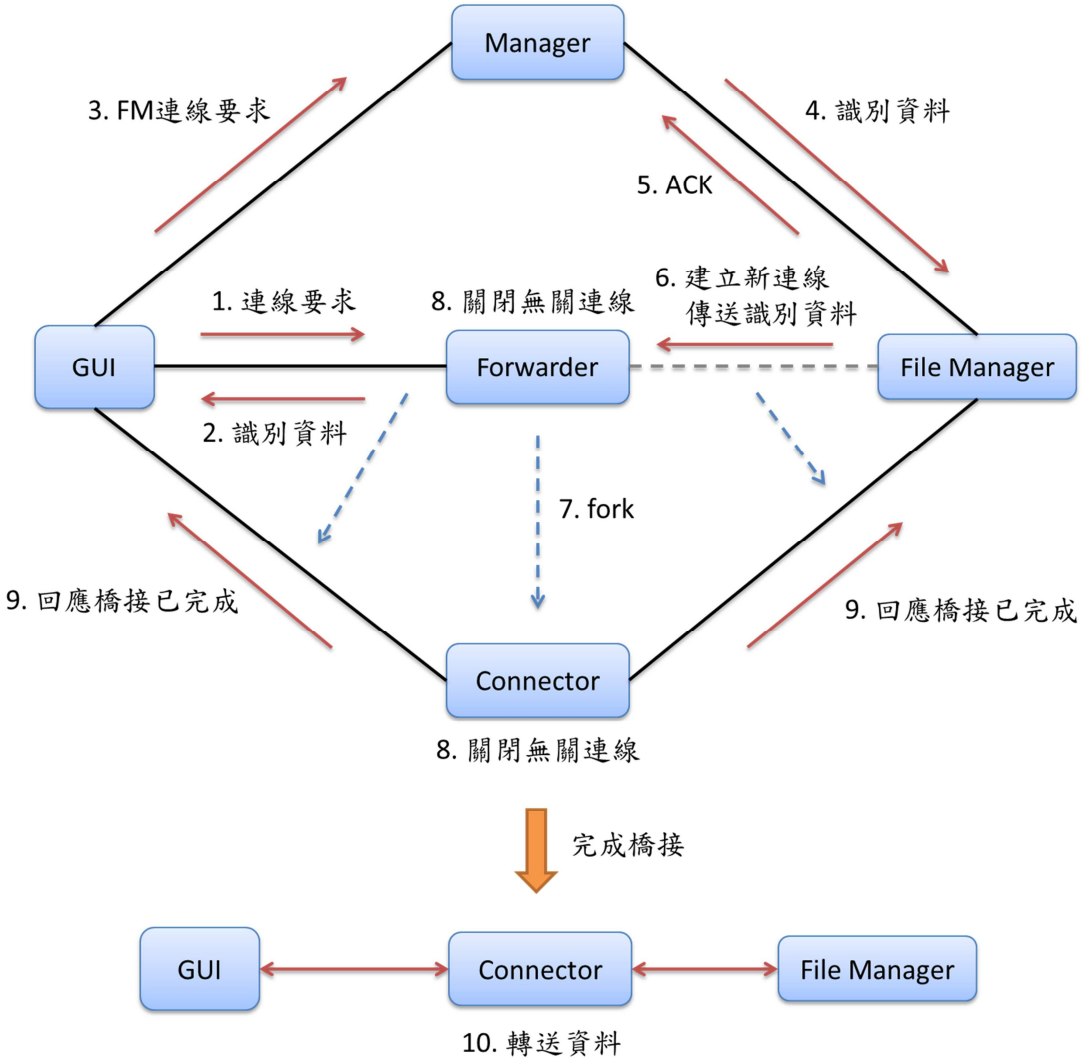

圖 4-2、GUI 與 File Manager 透過 Forwarder 建立連線

雲端系統使用者的帳號及密碼儲存在資料庫中，我們以 MySQL 作為資料庫。使用者透過 GUI 傳送帳號及密碼，透過上述之 Forwarder 轉送機制傳送至 Manager。Manager 以資料庫的資料驗證收到的帳號密碼是否正確，如正確則記錄此連線由哪個使用者所使用，代表此連線已通過認證之連線。GUI 透過連線傳送各種要求給 Manager 時，Manager 會檢查該連線是否已通過認證，是則執行相應工作，否則拒絕 GUI 的要求。而 GUI 與 File Manager 連線的建立則需透過 Manager，Manager 只會為已通過認證的 GUI 建立此連線，因此 File Manager 可不須再認證 GUI。

4.2.2 傳送模擬工作

使用者登入雲端系統、完成網路拓撲後，即可傳送模擬工作至雲端系統。在傳送模擬工作時，使用者須設定模擬工作名稱、此工作要用多少虛擬機器同時進行模擬、網路參數變化的設定、選擇要使用的模擬引擎。

網路模擬中網路拓撲主要由許多節點及節點間的連線組成。這些節點可能是主機（host）、Switch、Router、無線 AP（Access Point）、擁有無線傳輸能力的主機等。連線則可能是有線網路或各種無線網路，如 802.11n。

在 EstiNet 網路模擬器中，每個節點有一個或多個「Port」，此處 Port 是指節點上的網路介面，而非 TCP、UDP 協定中的 Port 概念。每個 Port 上有一協定堆疊（Protocol Stack），協定堆疊由許多協定模組（Protocol Module）組成，如 FIFO、802.3 MAC。如背景中所述，協定模組是 EstiNet 網路模擬器中實際運作協定規則的元件，封包進出時會依序經過這些協定模組的處理。在協定模組中有許多參數可供使用者調整，如 FIFO 模組的最大佇列長度（Maximum Queue Length）。

使用者在傳送模擬工作時以 GUI 設置他想變動的網路參數，GUI 將使用者設定網路參數的指令寫在一文字檔中，傳送模擬設定檔時一同送至雲端系統。以下說明該文字檔及設定參數的指令格式。

網路參數設定檔格式如下：

```
<Name>,<Setting>
<Name>,<Setting>
……
```

每一行代表一個參數變動的設定，使用者可同時設定多個參數。<Name>表示參數名稱，<Setting>表示參數值的設定，兩者以逗號隔開，詳細說明如下。

<Name>有兩種格式，分別用於指定全域變數及節點協定模組中的參數：

1. G,<Global Variable Name>
   此格式用於指定全域變數的名稱。
   <Global Variable Name>為欲修改的全域變數名稱，如 RandomNumberSeed。

例如：G,RandomNumberSeed

2. N,<Node ID>,<Port ID>,<Module Name>,<Module's Variable Name>
此格式用於指定節點協定模組中參數的名稱。
<Node ID>為欲設置節點的 ID。
<Port ID>為欲設置 Port 的 ID。
<Module Name>為欲設置參數所在模組名稱。
<Module's Variable Name>為欲設置之參數名稱。
例如：N,1,2,FIFO,max_qlen，表示要設定節點 1 的 Port 2 模組堆疊中 FIFO 模組的最大佇列長度（Maximum Queue Length）。

在參數值<Setting>的設定上，我們提供兩種格式：

1. R,<min value>,<max value>,<delta>
此以等差數列的方式設定想要的參數值，以上三個參數分別代表最小值、最大值及變動值。
例如：R,10,100,10 表示要設定的參數值為 10、20、30、…、90、100。

2. V,<Value 1>,<Value 2>,<Value 3>,...
此格式讓使用者可任意定義參數值。
例如：V,2,6,7,12,3,20 表示要設定的參數值分別為這六個數值。

在設置網路參數變化上，使用者可以指定多個參數變化，例如：

```
G,RandomNumberSeed,R,1,3,1
N,2,1,FIFO,max_qlen,R,10,50,10
```

以上設定表示亂數種子為 1 到 3，節點 2 的 Port1 的 FIFO 模組中最大佇列長度為 10、20、30、40、50。依此設定，雲端系統會組合兩個參數值，產生 15 個模擬子工作。其中第一個模擬子工作的亂數種子為 1、最大佇列長度為 10，第二個子工作亂數種子為 1、最大佇列長度為 20，……，第七個子工作亂數種子為 2、最大佇列長度為 20，依此推類。

另外，為了讓參數設定更方便、有彈性，讓使用者可指定所有節點、節點上所有 Port 或指定所有相同類型節點，在<Node ID>及<Port ID>的設定上除了以數

字指定外亦可用「*」指定。各種設定方式及意義如表 4-1 所示，其中 N 及 M 為整數。

Node ID	Port ID	意義
N	M	指定節點 N 的 Port M
N	*	節點 N 的所有 Port
*	M	所有節點的 Port M
*	*	所有節點的所有 Port
N*	M	所有跟節點 N 類型相同之節點的 Port M
N*	*	所有與節點 N 類型相同之節點的所有 Port

表 4-1、節點及 Port 之設定意義

使用者完成雲端系統相關設定後傳送工作。GUI 會將此模擬工作的模擬設定檔，以及網路參數設定檔打包成一個檔案。接著從 EstiNet 認證伺服器（License Server）取得「雲端憑證（Cloud Certificate）」，與打包好的模擬設定檔一併傳送至雲端系統。雲端憑證將於章節 4.6 說明。

設計上，我們將作為控制中心的 Manager 跟處理檔案傳輸的 File Manager 分開，因此在傳送模擬工作時需要 File Manager 及 Manager 彼此溝通才能完成模擬工作的接收。我們用模擬設定檔的 SHA1 值讓 File Manager 識別 GUI 傳送的模擬設定檔屬於哪個模擬工作。雲端系統接收模擬工作步驟如下：

1. GUI 傳送模擬工作要求（request）的命令給 Manager，命令中包含模擬設定檔的 SHA1 值、雲端憑證等資訊。Manager 收到命令後將 SHA1 值、雲端憑證連同模擬工作的資訊存入資料庫。

2. GUI 將打包後的模擬設定檔傳給 File Manager。File Manager 以收到的檔案計算 SHA1 值，以此值至資料庫查詢、取得模擬工作 ID。

3. File Manager 告知 Manager 設定檔已就緒的模擬工作 ID，之後 Manager 可開始分配此工作。

模擬工作的設定檔會放在使用者於 NFS 上的暫存目錄，之後 Dispatcher 及模擬引擎皆透過掛載來存取模擬設定檔。

4.2.3 分配模擬工作

在 EstiNet 網路模擬器原本的架構中，一個 Dispatcher 可服務多個 GUI、處理多個模擬工作。但在雲端系統中，由於一個模擬工作會再產生多個模擬子工作，且每個模擬子工作皆可獨立執行（可視為原有架構中的一個模擬工作）。為了簡化 Dispatcher 的工作，我們以一個 Dispatcher 處理一個模擬工作，也就是一個 Dispatcher 處理一個模擬工作的所有模擬子工作。一個模擬工作的所有子工作皆完成時，代表該模擬工作完成。如果一個使用者有多個模擬工作，則每個模擬工作皆有一個 Dispatcher 處理。

Manager 是整個雲端系統的控管中心，負責分配 Dispatcher 及 Coordinator。Dispatcher 及 Coordinator 啟動時皆會連向 Manager，因此 Manager 知道目前系統中有哪些 Dispatcher、Coordinator 可供分配。

在 Manager 收到模擬工作要求後，會先檢查系統中是否有空閒的 Dispatcher 及 Coordinator，若有則以空閒的 Dispatcher 及 Coordinator 提供服務，若無足夠空閒的 Dispatcher 或 Coordinator 則開啟新的虛擬機器，即 DVM、CVM，以服務此模擬工作。如圖 4-3 所示：

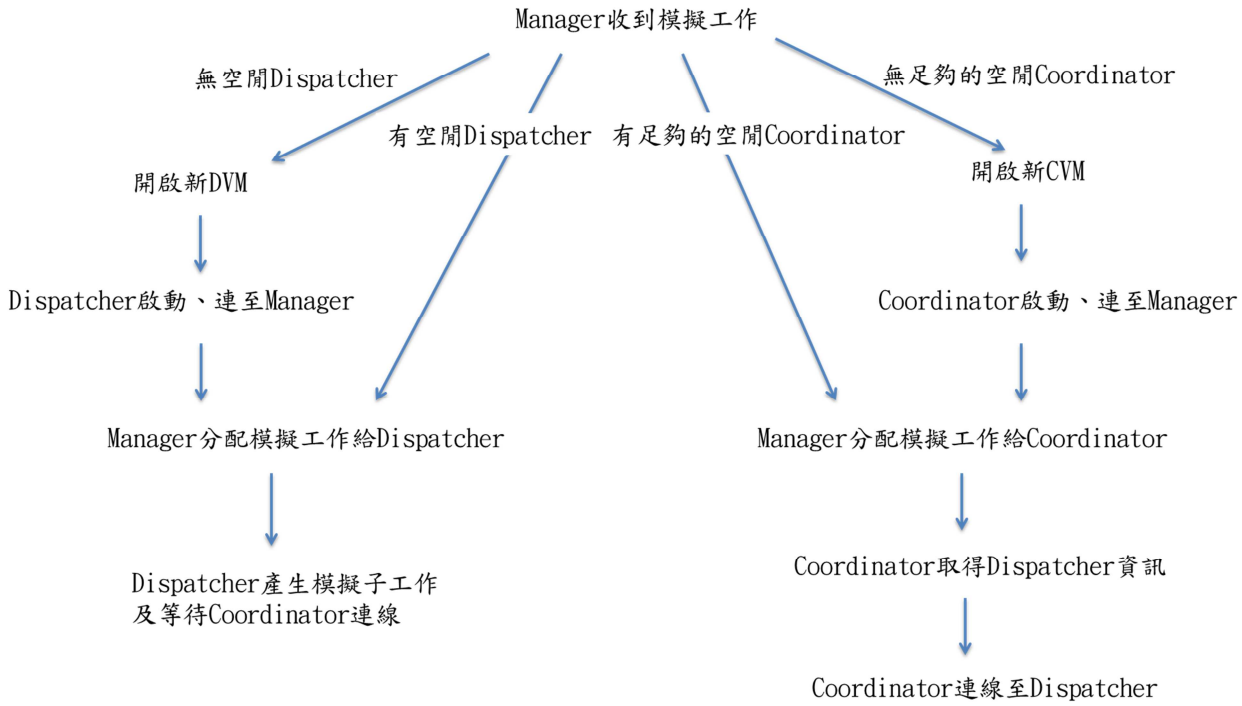

圖 4-3、分配模擬工作流程

Manager 分配 Dispatcher 給一個模擬工作的方式是通知 Dispatcher 模擬工作的相關資訊、要求 Dispatcher 開始處理模擬工作。在 EstiNet 原有架構中，Coordinator 於啟動時讀取設定檔、得知 Dispatcher 所在 IP Address，接著連向 Dispatcher 並向其註冊，Dispatcher 之後便可管理模擬機器。但在雲端系統中有多個 Dispatcher，且無法以固定設定檔設定 Dispatcher 的 IP Address。我們修改原有的連線機制，讓 Manager 於分配 Coordinator 時將負責該模擬工作的 Dispatcher 的 IP Address 告知 Coordinator，Coordinator 則可再自行連向 Dispatcher。

另外，Manager 開啟及分配虛擬機器時，會要求 NFS Exporter 分享相關目錄給這些虛擬機器，此將於章節 4.4 說明。Manager 分配完模擬工作後，會將模擬工作及負責虛擬機器的對應記錄在資料庫中。

Manager 開啟虛擬機器有兩個步驟：

1. 以事先建立的硬碟映像檔及 flavor 開啟虛擬機器。

2. 將一個未使用的 Floating IP Address 關聯至新啟動的虛擬機器。

在此雲端系統的建置中，須先手動在虛擬機器中安裝 EstiNet 網路模擬器需要的環境，接著以此虛擬機器的硬碟製作硬碟映像檔，供開啟虛擬機器時使用。如背景中所述，在 Openstack 中開啟虛擬機器須指定虛擬機器的運算資源，此設定稱為 flavor。我們在 Manager 中指定事先建立好硬碟映像檔及 flavor，用於開啟虛擬機器。

Manager 在資料庫中維護一份 Floating IP Address 表，記錄 Floating IP Address 跟虛擬機器的對應，代表哪台虛擬機器使用哪個 Floating IP Address。每開啟一台虛擬機器，Manager 會找一個沒被使用的 Floating IP Address，將之關聯到新開啟的虛擬機器，並更新資料庫、標示此 IP Address 由哪台虛擬機器使用。

由於 Floating IP Address 的關聯需在虛擬機器以從 Openstack 取得 Fixed IP Address 後才得以進行，因此「開啟虛擬機器」及「將 Floating IP Address 關聯至虛擬機器」兩個步驟之間會有一小段時間的間隔。

「開啟虛擬機器」及「將 Floating IP Address 關聯至虛擬機器」是使用 Openstack 的功能。如背景中所介紹，Openstack API 是一 RESTful Web 服務，我們透過傳送 HTTP 要求、接收 HTTP 回應使用 Openstack API。實作上我們使用 libcurl 函式庫（library）收送 HTTP 要求及回應。Manager 先以我們架設的 Openstack 系統的帳號及密碼取得認證 token，之後以 Openstack API 定義的指令及格式傳送 HTTP 要求，進行「開啟虛擬機器」、「將 Floating IP Address 關聯至虛擬機器」的操作。

在系統架構中提到，我們將虛擬機器分為專門執行 Dispatcher 的 DVM 及專門執行 Coordinator 及模擬引擎的 CVM。為達到雲端系統的自動化，我們需讓 Dispatcher 及 Coordinator 能自動啟動。因此我們將 Dispatcher 及 Coordinator 加入 Linux 開機時會啟動的程序中，讓在 Dispatcher 及 Coordinator 能在虛擬機器開啟後自動執行。我們依照 DVM 及 CVM 不同的需求撰寫啟動腳本（shell script），並將此腳本加入虛擬機器的/etc/rc.d/rc.local，Linux 於開機最後便會執行我們的啟動腳本。

在 Dispatcher 跟 Coordinator 間有兩條連線。一連線用於傳輸控制命令，如 Dispatcher 要求模擬機器開始模擬、Coordinator 註冊等控制命令皆透過此連線。另一連線用於傳輸模擬進度，此部分將於章節 4.2.7 說明。以下分別說明 Dispatcher 及 Coordinator 的啟動流程。

DVM 的啟動腳本執行以下動作：

1. 掛載（mount）NFS 上的目錄。

2. 執行 Dispatcher Manager。

為簡化軟體元件間的溝通及檔案傳輸，我們將模擬設定檔儲存在 NFS 中，讓 Dispatcher、Coordinator 及模擬引擎透過掛載 NFS 存取模擬設定檔。

DVM 中 Dispatcher 都是由 Dispatcher Manager 啟動。Dispatcher Manager 負責維持 DVM 中 Dispatcher 的個數，當一個 Dispatcher 完成工作、結束後 Dispatcher Manager 會再執行一個 Dispatcher。

實作上，Dispatcher Manager 先用迴圈 fork 出固定個數的子程序（child process），子程序以 execlp()執行 Dispatcher。接著用 sigsuspend()讓 Dispatcher Manager 暫停執行直到收到表示有 Dispatcher 結束的信號(signal)SIGCHLD 時，再 fork 子程序執行 Dispatcher。有多少 Dispatcher 結束 Dispatcher Manager 就會再 fork 多少子程序，讓一台 DVM 維持一定數量的 Dispatcher。

Dispatcher 啟動後的執行流程如下：

1. 讀取設定檔。

2. 建立資料庫連線，用於將模擬工作的資訊更新至資料庫。

3. 以亂數產生兩個 Port Number 並監聽。

4. 連接 Manager。

5. 向 Manager 註冊，將 IP Address 及控制用監聽 Port 告知 Manager。

6. 等待 Coordinator 的連線或來自 Manager 或 Coordinator 的控制命令

Dispatcher 完成上述啟動流程後，進入可接收模擬工作的狀態。其中會有兩個監聽 Port 等待 Coordinator 之連線。一用於建立控制用連線，一用於建立傳送模擬進度用連線。第 3 步中須以亂數產生 Port Number 是因一台 DVM 中會有多個 Dispatcher，無法以固定 Port 監聽。

CVM 的啟動腳本執行以下動作：

1. 設定執行 EstiNet 網路模擬器所需要的環境變數。
2. 關閉防火牆 iptables。
3. 執行 Fip Asker，若 Fip Asker 執行成功且正常結束則執行 Coordinator。

前兩個步驟為以 EstiNet 網路模擬器運行模擬時所需要的步驟，需要第 3 步驟之原因將於後說明。

Coordinator 啟動後的執行流程如下：

1. 初始化變數、產生相關物件，設定環境變數及讀取設定檔。
2. 連接 Manager。Manager 的 IP Address 及監聽 Port 由第一步讀取設定檔時取得。
3. 等待 Manager 的分配命令。此命令帶有這個 Coordinator 被分配給哪個模擬工作、由哪個 Dispatcher 管理的資訊。
4. 收到 Manager 命令後從中取得 Dispatcher 的 IP Address、Port 以及此模擬工作所屬的使用者。
5. 掛載該使用者在 NFS 上用以存放模擬設定檔及結果檔的目錄。
6. 以第四步取得的資訊連接 Dispatcher，此為 Coordinator 及 Dispatcher 間用於傳送控制命令的連線。

7. 向 Dispatcher 註冊，從 Dispatcher 收到另一個監聽 Port Number。

8. 以第七步取得的資訊與 Dispatcher 建立第二條連線，此連線用於傳送模擬進度，於章節 4.2.7 詳細說明。

Coordinator 完成上述啟動流程後，進入可開始進行模擬的狀態。

在我們的系統架構中，虛擬機器存取 NFS 時，NFS 看到的虛擬機器 IP Address 是 Openstack 架構中的 Floating IP Address，因此在 NFS 的設定上是以 Floating IP Address 進行目錄的分享設定。在前述 Manager 啟動虛擬機器的流程中，開啟虛擬機器到虛擬機器擁有 Floating IP Address 間會間隔一小段時間，此段時間會隨著虛擬機器開啟數量的增加而增長。

Coordinator 啟動後會連向 Manager，如果此時 CVM 尚未擁有 Floating IP Address，Manager 以 getpeername()取得連線對方的 IP Address 時無法取得 CVM 的 Floating IP Address，會由於 Floating IP Address 的機制而得到 Controller 的 IP Address。這會造成 CVM 無法掛載 NFS 或對應錯誤使 Manager 無法識別此台 CVM。

有此問題是因為 Coordinator 在 CVM 尚未取得 Floating IP Address 時便連向 Manager。我們希望 Coordinator 是在已有 Floating IP Address 時才連向 Manager，因為 Floating IP Address 的機制，我們無法從 CVM 中得知自己的 Floating IP Address 再傳送給 Manager。最後我們以 Fip Checker 跟 Fip Asker 解決此問題，以下說明解決方式。

在 Worker 上執行 Fip Checker，功能為檢查 IP Address 是否為 Floating IP Address。在 CVM 啟動後且 Coordinator 執行前，我們會先執行 Fip Asker 用於確認虛擬機器已取得 Floating IP Address。

Fip Checker 查詢 Openstack 資料庫中資料表 floating_ips 中欄位 deleted 為 0 的 IP Address，得到目前 Openstack 系統中使用的 Floating IP Address 範圍。Fip Asker 啟動後會連接 Fip Checker，Fip Checker 用 getpeername()取得 Fip Asker 的 IP Address，比對是否為 Floating IP Address，是則回傳 OK 後關閉連線，若不是

Floating IP Address 則直接關閉連線。Fip Asker 如果偵測到 Fip Checker 直接關閉連線則呼叫 sleep()睡眠一秒後再重新連線，Fip Asker 會不斷嘗試連線直到收到 OK 則結束執行。

在 CVM 的啟動腳本中，我們讓 Coordinator 在 Fip Asker 正常結束時啟動，因此當 Fip Asker 收到 OK 時正常結束即可讓 Coordinator 得以啟動。如此便可達到讓 Coordinator 在 CVM 擁有 Floating IP Address 的狀況下向 Manager 建立連線。

4.2.4 產生及分配模擬子工作

Dispatcher 收到 Manager 的模擬工作要求後會做一些簡單處理，接著依據變數設定產生可各自獨立執行的模擬子工作，最後將模擬子工作分配給模擬機器。以下說明前述各步驟的設計與實作。

首先，Dispatcher 解開打包的模擬設定檔，將模擬設定檔放在此工作所屬使用者於 NFS 上的暫存目錄，模擬引擎讀取模擬設定檔會從掛載 NFS 的目錄讀取。接著，Dispatcher 從資料庫中查詢此模擬工作的「雲端憑證」，用於分配模擬子工作時傳送給 Coordinator，此認證於章節 4.6 說明。

Dispatcher 依據使用者的參數變化設定產生模擬子工作。首先依據使用者參數變化的設定，組合出所有變數值的組合。接著依照組合的數量，產生相應數量的模擬子工作，並為每個子工作產生一物件記錄此工作之相關資訊。再從模擬工作的設定檔複製出多份設定檔、分別放到各子工作的目錄，模擬引擎進行模擬時便會到該子工作的目錄讀取設定檔。最後 Dispatcher 修改每個模擬子工作的設定檔，將要改變的網路參數寫入模擬描述檔（.tcl 檔）。

模擬描述檔會描述網路拓撲中，節點使用的協定模組之間的關係（即協定堆疊），以及拓譜中的節點的連接關係。模擬描述檔中亦會指定各種網路參數，如亂數種子（random number seed）等全域變數，以及節點中網路協定模組的參數值。此檔案於使用者以 GUI 繪製網路拓撲後，由 GUI 產生。模擬描述檔有自訂格式及指令，模擬引擎中有對應各指令的處理函式（function）。執行模擬時模擬引擎會讀取此檔案，依其中的指令建立物件、設置變數值，建立出各節點的協定

堆疊。模擬引擎讀取完整份模擬描述檔後，各節點協定堆疊的相關物件及關連便建立完成，之後模擬引擎便可以這些物件開始模擬。因此我們透過修改模擬描述檔，讓模擬子工作得以不同的網路參數進行模擬。

實作上，Dispatcher 會在模擬描述檔最後、RUN 指令前加入 CloudSet 指令，用來修改全域變數及節點中網路協定模組的參數值。由於 CloudSet 指令在模擬描述檔的最後，模擬引擎處理完前面的指令、要處理 CloudSet 時，所有網路模擬物件皆以建立，此時便可直接修改模擬物件中的變數值，達到修改的目的。

CloudSet 指令分成兩種格式，分別修改全域變數與修改各節點各項變數，其格式與意義說明如下：

1. CloudSet <Global Variable Name> = <Value>
   此格式用於修改全域變數。
   <Global Variable Name>是全域變數名稱。
   <Value>是欲給予的值。

2. CloudSet Node <Node ID> Port <Port ID> <Module Name>.<Variable Name> = <Value>
   此格式用於修改節點協定模組中的參數值。
   <Node ID>為欲修改節點的 ID。
   <Port ID>為該節點中欲修改 Port 的 ID。
   <Module Name>為該 Port 上欲修改之協定模組名稱。
   <Variable Name>則為該模組中欲修改之變數名稱。
   <Value>為欲給予的值。

以下用兩個例子說明使用者於傳送工作時的參數設定如何轉換為 CloudSet 指令。

例一

使用者設定：

```
G,RandomNumberSeed,R,1,3,1
N,2,1,FIFO,max_qlen,R,10,50,10
```

以上設定表示亂數種子為 1 到 3，節點 2 的 Port1 的 FIFO 模組中最大佇列長度為 10、20、30、40、50。此設定會產生 15 個模擬子工作。

經 Dispatcher 組合變數值，並修改模擬子工作的模擬描述檔，於第一個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 1
CloudSet Node 2 Port 1 FIFO.max_qlen = 10
```

第二個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 1
CloudSet Node 2 Port 1 FIFO.max_qlen = 20
```

第七個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 2
CloudSet Node 2 Port 1 FIFO.max_qlen = 20
```

由例一可看到，Dispatcher 將節點 ID、Port ID、模組名稱、參數名稱套入 CloudSet 指令，參數值則是依照組合的結果設置。

例二

假設網路拓撲中有節點 1 有 3 個 Port。使用者設定：

```
G,RandomNumberSeed,R,1,3,1
N,1,*,Phy,Bw,V,10,100,1000
```

以上設定第一行如例一所述，第二行表示將所有連線的網路頻寬（bandwidth）分別設為 10、100、1000 Mbps。此設定會產生 9 個模擬子工作。

經 Dispatcher 組合變數值，並修改模擬子工作的模擬描述檔，於第一個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 1
CloudSet Node 1 Port * Phy.Bw = 10
```

第二個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 1
CloudSet Node 1 Port * Phy.Bw = 100
```

第六個模擬子工作的模擬描述檔會加上：

```
CloudSet RandomNumberSeed = 2
CloudSet Node 1 Port * Phy.Bw = 1000
```

由例二可看到，Dispatcher 僅負責參數變化的組合，對於未指明 ID 的節點及 Port 不會加以展開，保留「*」並將此部分的展開交由模擬引擎處理。

此是由於我們希望分析模擬描述檔的程式僅在整體系統中的一個軟體元件上，即模擬引擎，避免模擬描述檔未來若修改格式需維護多份分析程式。Dispatcher 不會對模擬描述檔進行分析，無法知道網路拓撲中有哪些節點及 Port。所以在使用者未明確指定節點或 Port 時，Dispatcher 無足夠資訊將「*」轉為多個指明節點及 Port 的 CloudSet 指令。如例二中 Dispatcher 對於第六個模擬工作無法產生以下指令：

```
CloudSet RandomNumberSeed = 2
CloudSet Node 1 Port 1 Phy.Bw = 1000
CloudSet Node 1 Port 2 Phy.Bw = 1000
CloudSet Node 1 Port 3 Phy.Bw = 1000
```

在模擬引擎中以類別（class）實作模組，這些類別會繼承 NslObject，此處的模組泛指協定模組（protocol module）及節點模組。在模擬引擎的實作上，將「節點」也視為模組，主要用於記錄節點資訊。而協定模組則是協定堆疊（protocol stack）中協定的實作，主要為實現協定之功能。

網路拓撲中的節點擁有一個或多個協定堆疊（每個 Port 都有一個協定堆疊），協定堆疊中有許多協定模組物件，因此每個節點擁有許多協定模組物件。這些物件由協定模組的類別產生，於模擬時扮演真正執行功能的角色。在模擬引擎中將一個節點擁有的協定模組物件串成一個串列。

由於模組皆繼承 NslObject，因此在做各種節點模組搜尋、協定模組搜尋時，皆可使用 NslObject 指標存取不同模組。

我們在模擬引擎中加入處理 CloudSet 指令的函式（function）。模擬引擎自模擬描述檔讀取 CloudSet 指令時會修改各項參數值，其實作方式如下。

在類別（class）CmdProcessor 增加 cmdCloudSet()函式，用以處理 CloudSeet 指令。修改 Command_Dispatch()，使之能認得 CloudSet 為一合法指令並呼叫 CmdProcessor 的 cmdCloudSet()指令。

修改全域變數直接使用模擬引擎中已有的函式 GetVariableBinder()->setVarValue()設定，呼叫此函式的參數為一 NULL 的 NslObject 指標、全域變數名稱、欲賦予的值。此函式在指定的 NslObject 指標為 NULL 時表示設定全域變數，NslObject 為模擬引擎架構中的基本模組類別，所有協定模組的類別皆須繼承此類別。

修改節點的協定模組參數，有以下步驟：

首先分析節點 ID 及 Port ID，區分 N、N*、*，接著依照不同組合有不同的處理。

1 如果節點 ID 包含數值（N 或 N*），取得此 ID 的節點物件。

1.1 若節點 ID 是 N*，表示要設置所有跟節點 N 相同類型的節點。以節點 ID 由小到大搜尋所有節點，找到跟節點 N 類型相同的節點，再看是否指定 Port。

1.1.1 有指定 Port 則設定該節點的模組變數。

1.1.2 無指定 Port 則搜尋此節點上所有與欲設置模組相符的模組設置變數。

1.2 若節點 ID 是 N，表示設置某個節點的模組變數。看是否指定 Port。

1.2.1 有指定 Port 則設定該節點的模組變數。

1.2.2 無指定 Port 則搜尋此節點上所有與欲設置模組相符的模組設置變

數。

2　若節點 ID 為*，表示要設定所有節點。

2.1　以節點 ID 由小到大搜尋所有節點，並以節點 ID 取得節點物件。

2.2　有指定 Port 則設定該節點的模組變數。

2.3　無指定 Port 則搜尋此節點上所有與欲設置模組相符的模組設置變數。

在模擬引擎中，一個節點上所有 Port 的協定模組是串在一個串列中，每個協定模組會記錄它在哪個 Port 上，但無資料結構記錄一個節點有哪些 Port。因為要設置節點上所有 Port 的 A 模組等同要設置此節點上所有的 A 模組，所以我們只須搜尋整個模組串列，找出我們要設置的模組 A 再加以設置變數。

在變數設置的實作上，先以節點 ID、Port ID、模組名稱為參數呼叫 InstanceLookup()取得模組物件，再使用 GetVariableBinder()->setVarValue()以模組物件指標、變數名稱、變數值作為參數設定模組變數。

除了產生模擬子工作外，Dispatcher 另一個主要工作為分配子工作。Dispatcher 在特定事件發生時才進行子工作的分配，如此可避免不斷檢查可否分配子工作的額外負擔。Dispatcher 在以下三個時機會嘗試分配子工作給模擬機器：

1　產生完模擬子工作後，因為如果此時已有模擬機器則應開始進行模擬。

2　Coordinator 註冊時，因為此時表示有空閒的模擬機器。

3　有模擬機器的狀態由忙碌轉為空閒時，因為此時表示有空閒的模擬機器。

Dispatcher 分配子工作給模擬機器時，會要求 Coordinator 設置執行模擬的環境。Coordinator 會設置工作目錄（work directory），此目錄之後會成為模擬引擎的工作目錄。在雲端系統中，工作目錄即為該模擬子工作設定檔所在的目錄，其位於 NFS 中。

4.2.5 執行模擬

Coordinator 完成模擬環境設置後，Dispatcher 即要求它開始進行模擬。Coordinator 在啟動模擬引擎前，須做一些前置處理，主要目的有二：

1 讓模擬引擎可取得雲端憑證，認證之用途將於章節 4.6 說明。

2 讓使用者得以在雲端系統中使用自行開發的協定模組及應用程式進行模擬。

為達到第一個目的，Coordinator 自 Dispatcher 收到雲端憑證，將之存至一暫存檔，之後此檔案會由模擬引擎讀取。接著 Coordinator 設定三個環境變數：

- ESTINET_MODE：代表模擬器運行模式，原單機模式或雲端模式。
- ESTINET_CERTPATH：雲端憑證暫存檔路徑。
- ESTINET_CERTLEN：雲端憑證的長度。

由於模擬引擎在單機模式及雲端模式中會有部分差異，因此須以 ESTINET_MODE 環境變數告知模擬引擎要以哪個模式運作。ESTINET_CERTPATH 及 ESTINET_CERTLEN 是讓模擬引擎取得雲端憑證。

為達到第二個目的，Coordinator 會自使用者家目錄複製使用者上傳的模擬引擎及應用程式執行檔 EstiNet 網路模擬器中模擬引擎及應用程式預設的放置目錄。我們以此方式讓使用者得以在雲端系統中使用其自行開發的協定模組及應用程式進行模擬。

前置處理完成後，Coordinator fork 出子程序，子程序執行模擬引擎開始進行模擬。在 Linux 系統中子程序會繼承父程序（parent child）的環境變數及工作目錄（work directory）。

模擬引擎是 Coordinator 的子程序（child process），因此可取得 Coordinator 設置的環境變數，從中得知運作模式以及取得雲端憑證。如章節 4.2.4 所說明，

Coordinator 的工作目錄會在設置模擬環境時設於該模擬子工作模擬設定檔所在的目錄。模擬引擎的工作目錄會設於此目錄，之後模擬結果檔會產生在此目錄。

4.2.6 完成模擬及釋放資源

一模擬工作完成後，雲端系統需更新相關資訊、清除暫存檔、自動釋放此模擬工作佔用的資源，如虛擬機器及 Floating IP Address，讓資源可以重複使用、服務下一個模擬工作。

模擬引擎完成模擬時會通知 Coordinator，Coordinator 再通知 Dispatcher 子工作完成。Dispatcher 得知一模擬子工作完成時，會將模擬結果打包放到使用者的家目錄，更新資料庫中此模擬子工作的相關資訊，並刪除子工作的暫存資料夾。如果一個模擬工作所有的子工作皆完成，Dispatcher 會刪除整個模擬工作的暫存資料夾。

Dispatcher 分配完模擬子工作後，以下兩種事件出現時會釋放 Coordinator、通知 Manager 關閉 CVM：

1 模擬機器的狀態由忙碌轉為空閒。
此為一般模擬工作進入結束階段的狀況，即已無等待中的模擬子工作需要執行，可釋放空閒的模擬機器。

2 Coordinator 註冊。
造成此狀況的原因有二：一為 CVM 開啟較慢、模擬子工作較小，使得有 CVM 開啟時已無等待分配的模擬子工作。二為使用者指定了比模擬子工作總數多的 CVM 數量。

Dispatcher 釋放所有 Coordinator 後會通知 Manager 它已完成模擬工作，讓 Manager 可更新相關對應，之後 Dispatcher 會結束執行。我們的設計中希望 DVM 上能維持一定個數的 Dispatcher 個數，因此如章節 4.2.3 所述，當一個 Dispatcher 結束時會再由 Dispatcher Manager 新啟動一個 Dispatcher。

Manager 收到 Dispatcher 要求釋放 CVM 時，Manager 會先要求 Coordinator 結束執行。待 Coordinator 關閉與 Manager 間的連線後，Manager 使用 Openstack

API 關閉那台 CVM、卸離(disassociate)關聯在該台 CVM 上的 Floating IP Address，最後更新資料庫中相關的對應資料。

4.2.7 取得模擬資訊及結果

模擬工作傳送至雲端後，使用者可透過 GUI 取得相關資訊，如使用虛擬機器的數量、模擬工作的 ID、工作名稱、目前狀態、模擬進度等。這些資訊儲存在資料庫，分別由 Manager 及 Dispatcher 更新，並由 File Manager 於 GUI 要求時查詢並傳送給 GUI。Manager 在接收模擬工作時會更新模擬工作資訊，如工作名稱。Dispatcher 負責更新模擬子工作的狀態、模擬進度、開始執行模擬的時間以及完成時間。

為了避免過多軟體元件都包含更新資料庫的程式碼、降低程式維護上的難度，我們將模擬子工作的資訊更新集中至 Dispatcher。從模擬引擎得到的資訊，如模擬進度、錯誤訊息等，會經由 Coordinator 交給 Dispatcher，再由 Dispatcher 更新至資料庫。Dispatcher 分配子工作時會將該子工作的狀態由等待改為執行中，並記錄此時間為開始執行模擬的時間。在模擬進行中，Dispatcher 會從 Coordinator 收到目前模擬的進展，經計算後將模擬進度更新至資料庫，細節於此節後半部說明。當一個子工作完成時，Dispatcher 會將狀態從執行中改為完成。如果模擬引擎異常結束，造成子工作無法正確完成，Dispatcher 亦會從 Coordinator 得知此資訊，此時 Dispatcher 會將工作狀態改為失敗，同時將工作失敗的原因記錄至資料庫。

單機版中，模擬進度是在 GUI 要求時才透過 Coordinator 取得，因原有架構中 GUI 與 Coordinator 間有連線，GUI 可直接要求模擬進度。但在雲端系統中，GUI 及 Coordinator 不再有連線，無法以舊有方式觸發並取得模擬進度。

對此，我們希望能自動更新模擬進度到資料庫，而非在 GUI 要求時才更新，GUI 改透過 File Manager 取得資料庫中的資訊即可得到模擬進度。我們讓 Coordinator 每隔一段時間詢問模擬引擎目前模擬世界的時間點，此處模擬時間以

tick 為單位，tick 可依據設定對應到不同的時間長度。接著 Coordinator 將 tick 資訊傳送給 Dispatcher，Dispatcher 再計算模擬進度。

實作上我們以一變數記錄 Coordinator 程式主迴圈的執行次數，此迴圈為呼叫 select()等待連線為可讀，select()返回後處理相應的連線，之後依照收到的命令進行相關工作。當迴圈執行達一定次數時詢問模擬引擎目前的 tick 再傳送給 Dispatcher。

我們原先以 Dispatcher 及 Coordinator 間的控制連線傳送 tick 資訊，但如此實作造成 tick 資訊可能穿插在其他 IPC（Inter-Process Communication）指令中，因為網路不保證先送的指令一定會先到。我們實作上是以同步的方式處理各軟體元件間的溝通，tick 資訊穿插在其他命令中會造成錯誤。因此我們新建一條連線專門傳送 tick 資訊，以解決此問題並達到自動更新模擬進度的功能。

4.2.8 對模擬工作進行操作

使用者使用雲端模擬服務時，可能因為模擬工作設定錯誤等因素需要停止正在進行的模擬工作，對於已完成、取回結果、不需留存於雲端系統的模擬工作，使用者亦有刪除模擬工作的需求。因此我們提供使用者可停止正在進行模擬的子工作、刪除已完成的模擬工作的功能。

在停止進行中的模擬子工作部分，我們使用 EstiNet 網路模擬器原有的停止功能，並稍作修改。在原有架構中，由 GUI 傳送停止指令給 Coordinator 要求停止模擬，Coordinator 再要求模擬引擎停止模擬。但在雲端架構中，GUI 與 Coordinator 已無連線，無法以此管道要求停止模擬。我們讓 GUI 傳送停止某個模擬子工作的指令給 Manager，由 Manager 通知負責該模擬工作的 Dispatcher，再由 Dispatcher 傳送停止指令給負責該子工作的 Coordinator，最終 Coordinator 以原有方式要求模擬引擎停止模擬。

在刪除模擬工作的部分，雲端系統會刪除資料庫的相關資料以及使用者儲存空間中的模擬結果檔。此功能同樣由 GUI 送出要求，而由 File Manager 負責處理。

4.3 使用者自訂模擬引擎及應用程式

EstiNet 網路模擬器可供使用者自行開發或修改網路協定模組，使用者可依照模擬器提供的 API 進行開發，並編譯（compile）開發的模組到模擬引擎中，執行他所需要的相關網路模擬。

在 EstiNet 網路模擬器中可直接使用真實世界的應用程式於模擬世界中傳送資料，不須額外針對模擬器環境開發專門的傳輸程式，因此使用者可使用自己撰寫的應用程式或其他現有的應用程式進行模擬。

綜合以上，我們希望讓使用者可在雲端系統中使用他自行開發的網路模組、自行撰寫的應用程式或其他現有應用程式進行模擬，以提高使用者使用雲端模擬服務的興趣。

使用者需要以他的協定模組進行併行模擬時，可先在自己的電腦上開發及測試，之後再將模擬引擎的執行檔上傳至雲端，於傳送模擬工作時指定他要使用的模擬引擎，便可使用他的協定模組進行模擬。

我們讓使用者透過 GUI 管理模擬引擎及應用程式的執行檔，使用者可上傳、刪除執行檔、修改檔案說明。以上功能在雲端系統中以處理檔案及資料庫資料的 File Manager 負責，檔案會存在使用者於雲端系統的家目錄，於章節 4.4 說明。

使用者傳送模擬工作時選擇要使用的模擬引擎，該模擬工作的所有子工作會以指定的模擬引擎進行模擬。如使用者無特別選擇自行開發的模擬引擎，會以預設模擬引擎進行模擬。Coordinator fork 出子程序執行模擬引擎前，會先複製使用者家目錄中模擬引擎及應用程式的執行檔到 EstiNet 網路模擬器的安裝目錄中。Coordinator fork 出子程序會執行使用者選擇的模擬引擎。模擬時使用應用程式是執行安裝目錄中 tools 目錄下的執行檔，因此只要將使用者上傳的執行檔複製至相應資料夾即可讓模擬引擎使用。

如章節 4.2.2 所述，File Manager 需連接 Manager 及 GUI，分別傳送及接收命令或資料。實作上，因為 File Manager 負責檔案傳輸，相較一般控制流程會花

比較多時間，為了不讓一個 GUI 的檔案傳輸卡住其他 GUI 的操作，我們 fork 出子程序（child process）分別服務每個 GUI。File Manager 本身（即父程序）則負責處理 Manager 的連線。圖 4-4 為 File Manager 父程序及子程序示意圖，其中的線段代表程序（process）間的連線：

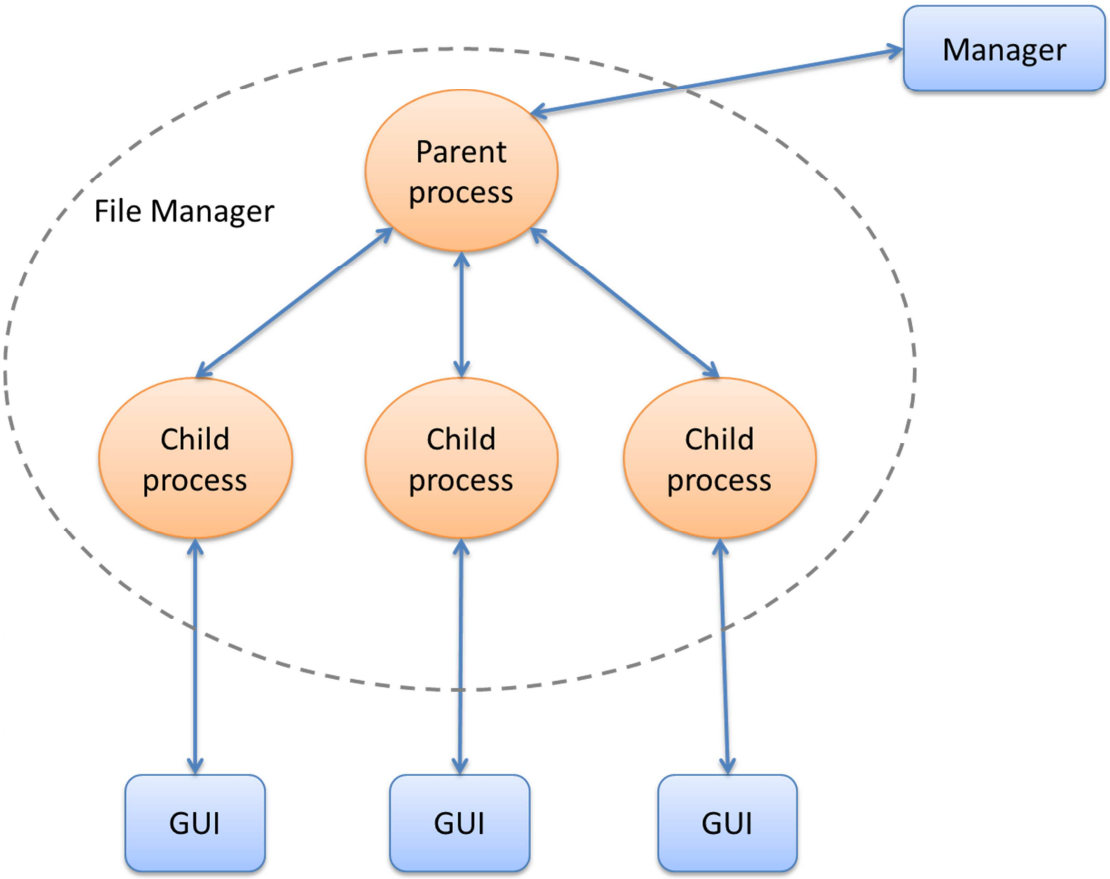

圖 4-4、File Manager 父程序及子程序

如章節 4.2.2 所述，使用者傳送模擬工作時，File Manager 及 Manager 需要一些溝通已完成模擬工作的接收。由於 File Manager 接收模擬設定檔時是以子程序接收，而收到模擬設定檔後需要告知 Manager 某個模擬工作的設定檔已經完成接收。

上述情境代表 File Manager 在接收完 GUI 的模擬設定檔後，必須通知 Manager 才算完成「接收模擬設定檔」這個動作。在以父程序服務 Manager、子程序服務 GUI 的情況下，父程序與子程序在接收模擬設定檔的這個動作中需互

相溝通，子程序需通知父程序它接收到一個模擬設定檔，再由父程序通知 Manager。

在 Linux 系統中，一台主機中的程序間可以多種方式進行 IPC（Inter-Process Communication），我們以建立 socket 連線解決此問題。如圖 4-4 所示，我們在父程序及子程序間建立一 socket 連線，子程序收到模擬設定檔後，便透過此連線傳輸命令通知父程序，父程序收到命令後再以它與 Manager 的連線通知 Manager。

4.4 儲存空間管理

在這個雲端系統中，我們以 NFS（Network File System）作為使用者存放各種檔案的儲存空間。

每個使用者在 NFS 上有兩個目錄，一為家目錄，一為暫存目錄。家目錄中存放使用者的模擬結果、自行開發的模擬引擎及應用程式執行檔以及模擬引擎異常結束時產生的 coredump 檔。暫存目錄用於使用者執行模擬時存放設定檔及模擬結果，模擬完成後才將模擬結果檔打包、壓縮並移至使用者的家目錄。

我們在 NFS 上先用兩個根目錄區分上述兩種目錄，再在兩個根目錄中分別以使用者帳號名稱建立使用者專屬的目錄，目錄結構如下所示：

```
cloudhome/
      user1/
      user2/
      ……
cloudtmp/
      user1/
      user2/
      ……
```

DVM 跟 CVM 會掛載（mount）上述兩種目錄。

CVM 會掛載使用它的使用者的兩個目錄，如 user1 的 CVM 會掛載 cloudhome/user1 及 cloudtmp/user1。由於 CVM 中可能執行使用者自行開發的模

擬引擎及應用程式，基於安全性考量，我們限制 CVM 只能掛載屬於該使用者的目錄。此安全性考量將於下一個小節說明。

DVM 會掛載兩個根目錄，即 cloudhome 及 cloudtmp。因為一台 DVM 中會執行多個 Dispatcher 分別負責不同模擬工作，這些模擬工作可能分屬不同使用者，使一台 DVM 可能處理多個使用者的模擬工作。DVM 需掛載 NFS 目錄結構中的根目錄，才能存取各使用者的模擬設定檔。由於 Dispatcher 是我們雲端系統中的軟體元件，不像 CVM 有安全性問題，因此不須像 CVM 限制只能掛載使用者的目錄。

Dispatcher 產生模擬子工作時，會在上述使用者的暫存目錄下建立子工作的目錄，並將模擬設定檔放在子工作的目錄中。模擬引擎執行於 CVM，可存取子工作的目錄，從中讀取模擬設定檔以及寫入模擬結果檔。

我們將使用者儲存於雲端系統的檔案類型分為：模擬結果檔、模擬引擎執行檔、應用程式執行檔以及 coredump。這些檔案會分別存在使用者家目錄中的不同子目錄，資料庫中也會記錄這些檔案的相關資訊。

4.4.1 檔案權限控管

EstiNet 網路模擬器執行模擬引擎時需要 root 身分的權限，因此我們在雲端系統的虛擬機器中都是以 root 身分執行模擬引擎。由於使用者可自行開發模擬引擎並上傳至雲端系統中使用，這代表使用者的模擬引擎可以虛擬機器的 root 身分執行。由於使用者上傳的是執行檔，目前我們沒有相關機制可判斷一個執行檔是否為模擬引擎，此表示使用者可以虛擬機器的 root 權限執行任意程式，相當於使用者擁有虛擬機器的 root 權限。

在網路模擬雲端系統中有許多安全性議題，我們較在乎的是模擬結果檔，因此在安全性上以保護使用者的模擬結果為優先。我們希望避免惡意使用者透過其上傳的執行檔存取、修改或刪除他人的模擬結果。

為避免使用者存取他人的檔案，我們希望 CVM 掛載 NFS 時，使用者 A 的 CVM 僅掛載使用者 A 的目錄而不能掛載其他使用者的目錄，以限制 CVM 對 NFS 檔案的存取。NFS 分享目錄時可指定哪些 IP Address 或網域可掛載該目錄，我們以此設定限制每個分享目錄的分享對象（即是可掛載它的 CVM）。

因為 CVM 是每次分配模擬工作時才分配給使用者作為該次模擬工作的運算單位，一個使用者使用的 CVM 會不斷變換，對 NFS 來說可掛載一個目錄 CVM IP Address 會不斷更改。

例如，使用者 A 第一個模擬工作使用三台 CVM，IP Address 分別為 10.10.10.101、10.10.10.102、10.10.10.103，此時 NFS 需將使用者 A 的目錄分享給前述三個 IP Address。當使用者 A 的第一個模擬工作結束時，NFS 需取消這三個 IP Address 對於使用者 A 目錄的分享。如果使用者 A 再進行下一個模擬工作，使用 IP Address 分別為 10.10.10.105、10.10.10.110 的 CVM，NFS 又需再設置相應的目錄分享。

NFS 在目錄分享上一般是手動寫入設定檔再做分享（export），但由於有上述 IP Address 與分享目錄對應的變動，我們希望 NFS 能動態分享目錄給虛擬機器。因此我們撰寫 NFS Exporter，負責動態分享目錄給虛擬機器。

Manager 在啟動 DVM 時，會要求 NFS Exporter 將兩種目錄的根目錄分享給新開啟的 DVM。Manager 在分配模擬工作時要求 NFS Exporter 將使用者的兩個目錄分享給分配的 CVM。Coordinator 啟動後先由 Manager 告知其負責工作屬於哪個使用者，Coordinator 再掛載 NFS 相應的資料夾。模擬工作完成時，Manager 會要求 NFS Exporter 移除對完成工作的 CVM 的分享。

實作上，我們在 Worker 上架設 NFS，並於 Worker 中執行 NFS Exporter。NFS Exporter 會接收 Manager 的命令，在/etc/exports 加入或刪除目錄分享設定，執行 exportfs 重新分享目錄。

NFS Exporter 設定的 IP Address 是由 Manager 告知，而 Manager 是由與 Coordinator 的連線取得 CVM 的 IP Address。如架構中所述，由於執行 Manager

的 Worker 位於內層網路之外，Manager 會得到的 IP 為 CVM 的 Floating IP Address，所以 NFS 在 IP Address 的設置上是使用虛擬機器的 Floating IP Address。

4.4.2 空間配額

由於使用者在雲端系統中會使用儲存空間，我們不希望使用者毫無限制的使用雲端系統的儲存空間，我們希望能記錄使用者儲存空間的使用量及控管空間配額。

原先我們希望以 Linux 本身具有之空間配額（quota）管理功能控管使用者的空間配額。但此方式需使用 NFS 分享目錄時需針對不同 CVM 做帳號身分的轉換，並整合存於資料庫中的使用者帳號資訊與 Worker 本身的 Linux 帳號。由於此整合可能需以 LDAP（Lightweight Directory Access Protocol）進行，但整合資料庫中的帳號資料與 Linux 系統的帳號目前在此雲端系統中只有空間配額的功用。在整合的效用與複雜度的評估下，我們認為以 LDAP 做此整合的整體效益不大，因此以另外的方式控管空間配額及空間使用量。

此雲端系統並非直接開放使用者存取儲存空間，任何與檔案有關的操作皆須透過 GUI，且與檔案有關的操作皆以我們的軟體元件進行。因此，我們可在與檔案操作的動作中加入空間使用量計算及檢查是否超過配額。

與檔案操作有關的動作有：

1 完成模擬，將結果檔移至使用者家目錄

2 使用者刪除模擬工作、刪除模擬結果

3 上傳及刪除模擬引擎或應用程式

4 模擬引擎異常結束、產生 coredump

5 刪除 coredump

我們將空間配額及空間使用量記錄於資料庫。以上與檔案有關的動作分別在 File Manager 及 Dispatcher，我們在這些動作的前後加入空間使用量的計算並更新資料庫。在使用者傳送模擬工作、上傳模擬引擎及應用程式時檢查使用者剩餘的空間是否足夠，不足則不讓使用者進行這些動作。在產生模擬結果、coredump 時則不會檢查空間是否足夠，避免因空間限制造成使用者的模擬結果或錯誤資訊遺失，但仍會計算使用量。若因這些檔案造成使用者使用超過配額的空間，之後他需刪除檔案、騰出空間才可再次傳送模擬工作。

在 NFS 目錄結構中，每位使用者有一家目錄及暫存目錄。做此設計的原是避免受到 Linux 空間配額的限制而使模擬過程中無法再寫入檔案，因為在模擬過程中可能產生較大的模擬結果檔，如封包紀錄檔。雖然系統最終未使用 Linux 的空間配額管理，但經過打包及壓縮可減少結果檔的整體大小，這可使同樣的硬碟空間能容納更多使用者的結果檔案，且 Dispatcher 於模擬結束時進行打包及壓縮亦可較簡易的計算檔案大小及空間使用量，因此我們仍保留此項設計。

4.5 模擬引擎異常處理

4.5.1 模擬引擎異常結束之問題

在正常運作中，模擬引擎執行完模擬會正常結束執行，使用者可再次進行下一個模擬工作。但當模擬引擎程式有錯誤或協定模組撰寫不當，會造成模擬引擎異常結束且未正常釋放佔用的 Linux 系統資源。因有些系統資源只能被一程序使用，前一次模擬工作未正常釋放這些資源時，將無法再次啟動模擬引擎，導致使用者無法進行下一次模擬。

以單機模式運行 EstiNet 網路模擬器若模擬引擎發生上述錯誤，使用者可直接重新啟動電腦。但在雲端系統中的模擬引擎發生錯誤、無法再次正常執行時，使用者無法如單機模式中可直接介入、取得錯誤資訊並重新啟動機器，該模擬機器便會進入無法執行模擬的狀態。因此我們需要針對模擬引擎於虛擬機器中異常

結束時做相關處理，使虛擬機器可再次正常執行模擬，並通知使用者此模擬子工作發生錯誤、未正確完成。

模擬引擎異常結束可能是因為使用者在開發協定模組上的問題，開發者一般希望在程式錯誤時有相關錯誤資訊協助除錯。因此，我們提供使用者模擬引擎異常結束時產生的 coredump 檔案，方便使用者除錯。

4.5.2 處理方式

Linux 的程序（process）從 main()返回（return）或呼叫 exit()時表示程序正常結束（terminate normally）。程式開發者可以 atexit()向系統註冊一函式，使該函式於程序正常結束時會被呼叫。模擬引擎向系統註冊 EstiNet_exit()，使其於模擬引擎正常結束時清除佔用的系統資源。因此在正常運作下，模擬引擎完成模擬後會正常釋放其佔用之資源。

在 Linux 中，系統在某些狀況下會送系統信號（signal）給正在執行的程序，例如子程序（child process）結束時父程序（parent process）會收到 SIGCHLD 信號。程序收到信號時，會執行信號處理函式（signal handler）。程式開發人員可在程式以 signal()函式向系統註冊自行定義的信號處理函式。若程式無自行定義信號處理函式，則會執行系統預設的信號處理函式及預設動作，如結束程序。若程式自行定義信號處理函式則執行該處理函式。

Linux 中若程序有存取非法記憶體位置、運算錯誤（如除 0）等行為，系統會傳送系統信號（signal）給程序，並強制中止程序，此時程序為異常結束（terminate abnormally）。此狀況中 process 不會執行以 atexit()註冊的 function。因此未修改的模擬引擎在異常結束時不會執行 EstiNet_exit()，即無法正常釋放佔用資源。

我們為處理此問題，針對多數信號向系統註冊自行定義的信號處理函式。此信號處理函式的運作如下：

1　以本節後面所述的通知機制通知 GUI。

2　執行 EstiNet_exit()釋放資源。

3　將信號處理函式設回系統預設。

4　把收到的信號再丟回給自己以產生 coredump。

需要上述第 3 及 4 兩個步驟，是因為在自行定義信號處理函式的狀況下收到信號後，會無法產生 coredump 檔案。因此我們在釋放資源後將信號處理函式設回系統預設的處理函式，再將同樣的信號送給模擬引擎本身，藉此產生 coredump。

如此一來，無論模擬引擎因何種問題收到系統信號而異常結束，皆可正常釋放資源，便能解決未正常釋放資源而無法再次啟動的問題。

如章節 4.2.5 所述，模擬引擎由環境變數 ESTINET_MODE 得知要運行於單機模式或雲端模式。在兩種模式中，模擬引擎異常結束時會以不同方式通知使用者。

單機模式中，模擬引擎會傳送即時訊息（Runtime Message）給 GUI，GUI 會以一視窗顯示模擬引擎的訊息，此訊息內容為模擬引擎收到某個信號後異常結束、未完成模擬工作。

在雲端模式中，模擬引擎與 GUI 間已無傳送即時訊息之管道，因此無法使用即時訊息通知 GUI。另外 GUI 可於送出模擬工作後任何時間離開，使用即時訊息傳送亦不恰當。因此我們改將此資訊存於資料庫中供 GUI 查詢。以下說明雲端模式中，模擬引擎如何通知使用者異常結束、未正確完成模擬。

模擬引擎因收到系統信號而異常結束時，它會通知 Coordinator 它收到信號而結束且模擬工作未正常完成。Coordinator 等待模擬引擎結束執行後，將 coredump 檔移至使用者家目錄中相應資料夾，最後通知 Dispatcher 這個子工作未正常完成、模擬引擎收到哪個信號以及 coredump 的檔名。Dispatcher 將這些資訊記錄至資料庫中，之後 GUI 可透過 File Manager 查詢相關資訊、下載 core dump 檔。流程如圖 4-5 所示：

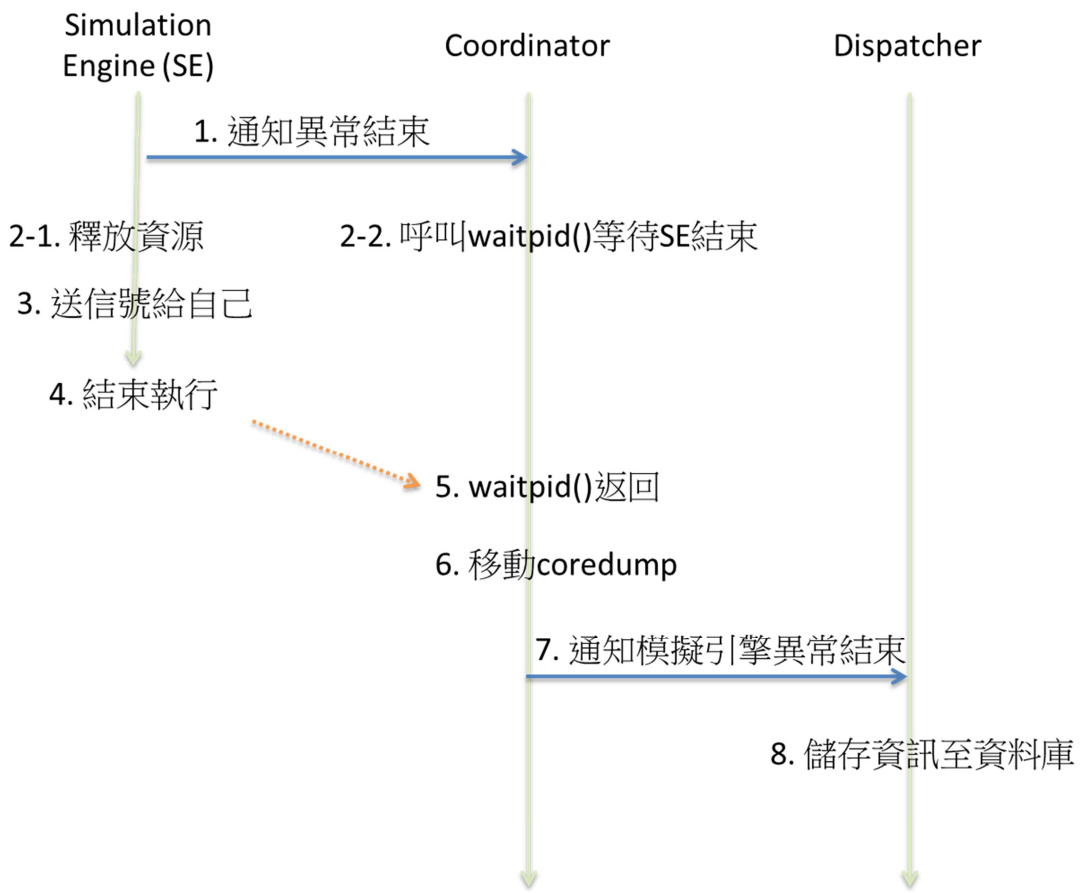

圖 4-5、模擬引擎異常結束處理

模擬引擎異常結束時，為讓使用者能取得異常結束的資訊以便使用者修正程式，我們將模擬引擎的 coredump 檔存在使用者家目錄中供使用者取得。

在 Linux 系統中，一般 coredump 會產生在程式所在的目錄。我們透過系統設定將 coredump 先統一產生於/tmp 目錄中，再由 Coordinator 將 coredump 移到使用者家目錄中的 file/coredump/目錄。以下說明系統設定。

Linux 系統中以 kernel.core_pattern 系統變數設定 coredump 檔案的產生位置，因此我們修改此系統變數更改 coredump 的產生位置。但由於 Fedora 預設上使用 ABRT（Automated Bug Reporting Tool）為套件軟體偵錯，而 ABRT 會使用 kernel.core_pattern。由於我們需修改此變數且並不特別需要 ABRT 之功能，用以下指令關閉 ABRT：

```
chkconfig abrtd off
service abrtd stop
```

我們用 sysctl 修改 kernel.core_pattern 變數的值：

```
sysctl -w kernel.core_pattern=/tmp/core
```

Linux 系統中另有一設置會影響程序異常結束時是否產生 coredump，此設置以檔案大小限制產生出的 coredump。若 coredump 檔案大小超過它的限制，就不會產生 coredump。我們希望不管 coredump 多大都會產生，所以將此設置設為不限制。設定方式為修改/etc/security/limits.conf，將其中 coredump 限制改為：

```
*                soft    core            unlimited
```

4.6 認證機制

EstiNet 網路模擬器是一個商業軟體，其中可以模擬各種網路，使用者須購買模擬能力才能模擬相關網路，如 VANET。因此商業機制，EstiNet 網路模擬器的 GUI 及模擬引擎啟動時會檢查使用者能使用哪些網路模擬能力。

我們的雲端系統中有一套僅用於雲端模擬服務的帳號系統，此帳號系統與上述模擬能力檢查、認證不同。模擬能力認證系統負責告知模擬器可使用哪些模擬能力，雲端帳號系統則負責雲端服務使用者的管理。這兩套認證系統是分開、獨立的。雲端服務使用者登入雲端時僅代表他可以使用雲端的併行模擬服務，但模擬器仍需經由模擬能力認證取得使用者可以使用的模擬能力。

以下說明雲端系統中的模擬能力認證機制。

4.6.1 原有認證機制介紹

EstiNet 網路模擬器原有的認證機制中，使用者購買模擬器後會得到一組 license key，安裝模擬器時會存至電腦中。之後 GUI 及模擬引擎啟動時，它們會連向認證伺服器（License Server）並以此 license key 嘗試認證、取得模擬能力。認證成功後認證伺服器會回傳模擬能力，代表可使用哪些網路協定進行模擬，例如可使用 802.11n、VANET 進行模擬。

在雲端架構中 GUI 位於使用者端，依然可連向認證伺服器。由於此雲端系統可能以私有雲（private cloud）的模式建立於非 EstiNet 公司之機構，基於資料安全性，不能將認證伺服器置於雲端系統內部。對虛擬機器來說認證伺服器處於雲端內部網路之外，而虛擬機器無法直接連接認證伺服器。虛擬機器中沒有使用者購買的 license key，與單機模式中已安裝在使用者電腦的情況不同。

綜合以上原因，虛擬機器中的模擬引擎無法透過原有的認證機制進行認證。我們需為雲端系統設計另一套認證機制，讓雲端系統中的模擬引擎能以使用它的使用者所擁有的模擬能力進行模擬。

4.6.2 雲端系統認證機制

因 GUI 可連向認證伺服器取得模擬能力資訊，我們希望透過 GUI 將此資訊傳給模擬引擎。模擬能力資訊是敏感資料，若有惡意使用者在傳輸過程中假造，則可非法使用模擬能力。因此除了要有讓模擬引擎取得模擬能力的機制外，我們還需要一些方式保護模擬能力資訊。

認證伺服器使用 RSA 加密演算法作為非對稱式加密的演算法，對模擬能力資訊進行加密及一些處理後產生「雲端憑證（Cloud Certificate）」。接著傳輸雲端憑證至模擬引擎，模擬引擎驗證並解開此雲端憑證、取得模擬資訊能力，之後進行模擬。由於 GUI 已經經過認證伺服器的認證且雲端憑證難以假造，模擬引擎可信任此模擬能力資訊。我們以此方式保護模擬能力資訊、避免惡意使用者假造，並讓模擬引擎可以得到使用者可使用的模擬能力。

以下說明實作細節。

首先說明認證伺服器產生雲端憑證的方式。認證伺服器以自己的私有金鑰（private key）加密模擬能力資料及模擬設定檔的 SHA1 值，此為 Message 部分。接著計算 Message 的 SHA1 值再以私有金鑰加密，此為 Signature 部分。Message 加上 Signature 即為我們的雲端憑證，此雲端憑證為二進位資料（binary data）。產生流程如圖 4-6 所示：

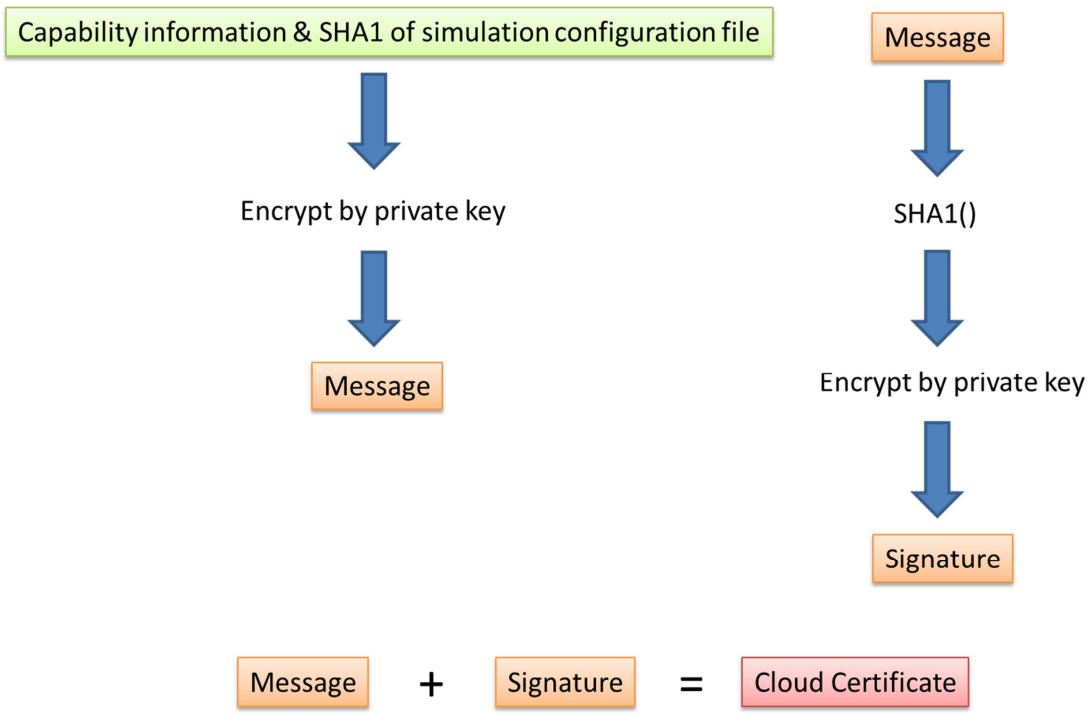

圖 4-6、認證伺服器產生雲端憑證

接著說明雲端系統如何驗證雲端憑證。雲端系統收到模擬工作時 Manager 會驗證雲端憑證的正確性，驗證成功後從雲端憑證取出模擬設定檔的 SHA1 值，與 GUI 於工作要求中所附的模擬設定檔 SHA1 值比對無誤後才是合法的模擬工作，若驗證失敗或比對 SHA1 失敗則丟棄此模擬工作。模擬引擎啟動時會驗證雲端憑證的正確性，驗證成功後從雲端憑證中取得模擬能力資訊進行模擬。圖 4-7 為驗證雲端憑證的方式：

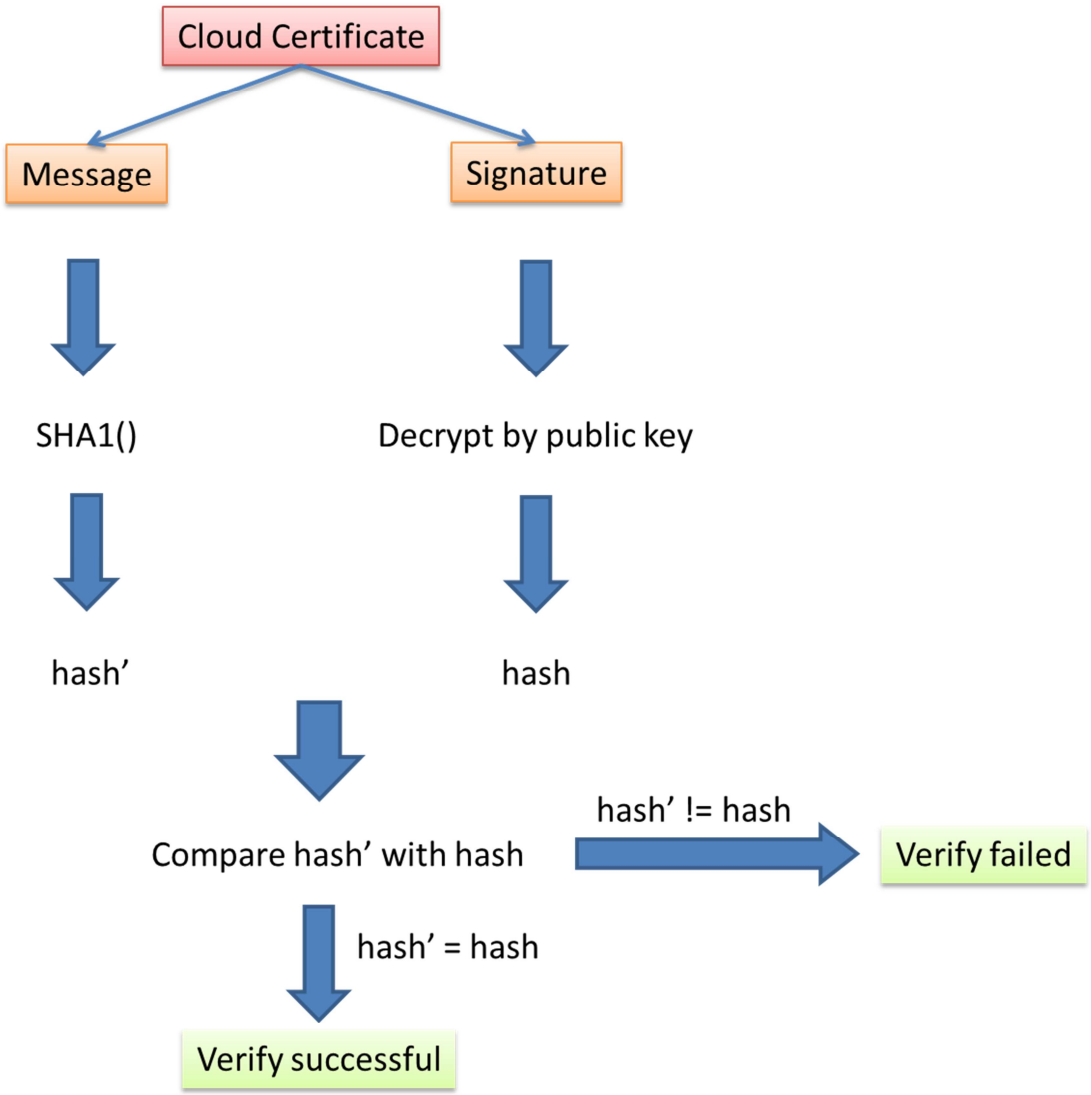

圖 4-7、雲端系統驗證雲端憑證

1 從雲端憑證取出 Message 及 Signature 兩部分。實作上我們在雲端憑證的前八個 byte 記錄 Message 及雲端憑證的長度，以此長度資訊區分出 Message 及 Signature。

2 計算 Message 的 SHA1 值，得到 hash'。

3 以公開金鑰解密 Signature，得到 hash。此為認證伺服器產生雲端憑證時計算出的 Message 的 SHA1 值。

4 比較 hash 及 hash'是否相同。如果相同則驗證成功，表示此為合法的雲端憑證。如果不同則驗證失敗，此憑證為非法。

如果雲端憑證驗證成功，則以認證伺服器的公開金鑰對 Message 解密，即可從 Message 中取出模擬能力資訊及模擬設定檔的 SHA1 值，如圖 4-8 所示：

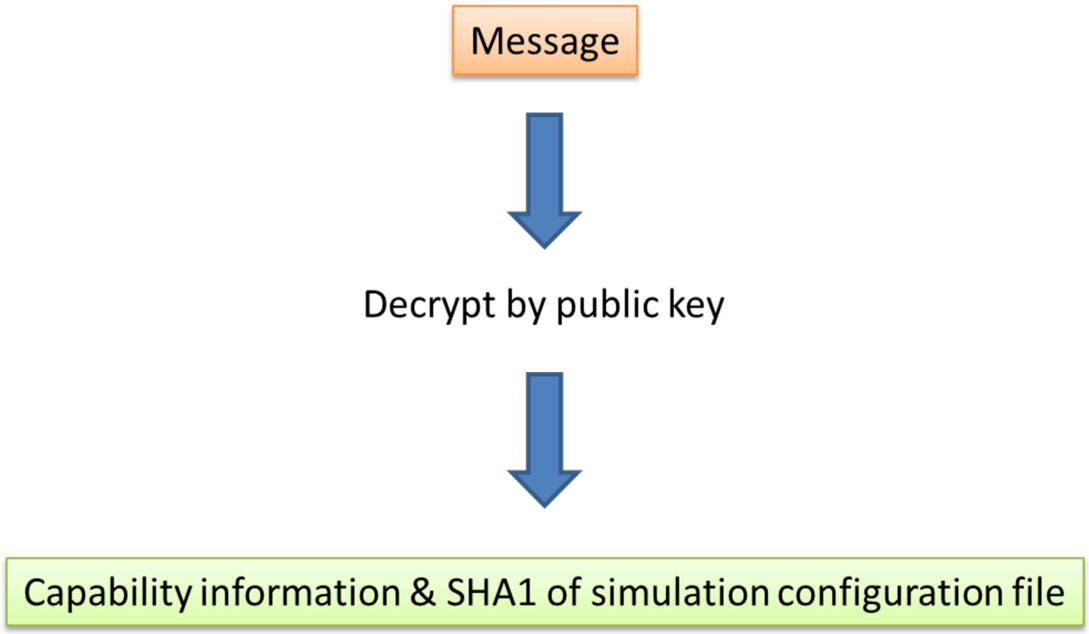

圖 4-8、由雲端憑證取得模擬能力

在非對稱式加密系統的觀念中，私有金鑰可用來識別一個人、組織或一台機器。因為私有金鑰應被擁有人妥善保管、僅擁有人會持有此金鑰，而公開金鑰則任何人皆可取得。若一段加密後的訊息可用公開金鑰解密，代表此段訊息必為私有金鑰持有人所加密。

在我們的認證機制中，認證伺服器擁有一私有金鑰，對應的公開金鑰則寫於模擬引擎及 Manager 的程式碼中，不完全對外公開。如果模擬引擎及 Manager 可用公開金鑰解開收到的 Signature，表示此 Signature 確實由認證伺服器加密。模擬引擎及 Manager 解密 Signature 得到一雜湊值（hash value），與自行計算 Message 的雜湊值進行比較，如果相符代表 Message 及 Signature 並未被竄改，且此雲端認證確實由認證伺服器發出。

我們的公開金鑰並未完全公開，但此處假設惡意使用者擁有認證伺服器的公開金鑰。以下以三種情境說明惡意使用者無法偽造雲端憑證。

情境一

惡意使用者可修改 Signature，使用公開金鑰解密修改過的 Signature 得到一偽造雜湊值。但因為雜湊函數的特性，惡意使用者難以變造出可計算出假造雜湊值的 Message。

情境二

惡意使用者可修改 Message 並計算出一偽造雜湊值，但他沒有認證伺服器的私有金鑰，無法產生對應的 Signature。

情境三

惡意使用者假造模擬能力資訊及模擬檔案 SHA1 值，但他沒有認證伺服器的私有金鑰，無法對假造資訊進行加密、產生偽造 Message。

另外，Message 本身為加密後資料，而原始資料有其格式，若模擬引擎解開 Message 後發現格式錯誤，亦可偵測 Message 是否有經過變造。

為了避免重播攻擊（replay attack），即使用者 A 攔截使用者 B 的雲端憑證，並以 B 的雲端憑證使用雲端模擬服務，我們希望為雲端憑證加入一些可供識別的資訊，降低此類攻擊的影響與可能。我們在此加入模擬設定檔的 SHA1 值，Manager 於時接收模擬工作時比對憑證中的設定檔 SHA1 值與工作要求所附的設定檔 SHA1 值是否相同，相同才視為合法。因為設定檔 SHA1 值難以假造，使每份雲端憑證僅能用於該次的模擬工作，因此重播攻擊無法讓惡意使用者使用別人的模擬能力進行自己想要的模擬工作。

本節最後說明雲端憑證如何從 GUI 傳送至模擬引擎。GUI 每次傳送模擬工作時，會以下列步驟取得雲端憑證並傳送至雲端系統提供模擬引擎認證：

1 GUI 與認證伺服器（License Server）互相認證對方是合法 GUI、認證伺服器。此步驟使用原有認證機制中的功能進行，在此不多加說明。

2 GUI 傳送 license key 及模擬設定檔的 SHA1 值給認證伺服器。

3 認證伺服器查詢認證資料庫，取得模擬能力資訊，依前述方式產生「雲端憑證」。

4 認證伺服器回傳雲端憑證給 GUI。

5 GUI 傳送模擬工作，將模擬設定檔連同雲端憑證傳送至雲端。

6 Manager 收到後將雲端憑證存於資料庫。

7 Manager 要求 Dispatcher 開始工作，Dispatcher 從資料庫取出雲端憑證。

8 Dispatcher 將雲端憑證傳給 Coordinator。

9 Coordinator 將雲端憑證存於一暫存檔中，並將暫存檔路徑存於環境變數。

10 Coordinator fork 出的模擬引擎由環境變數取得暫存檔路徑、讀取雲端憑證進行驗證。

上述傳遞如圖 4-9 所示。

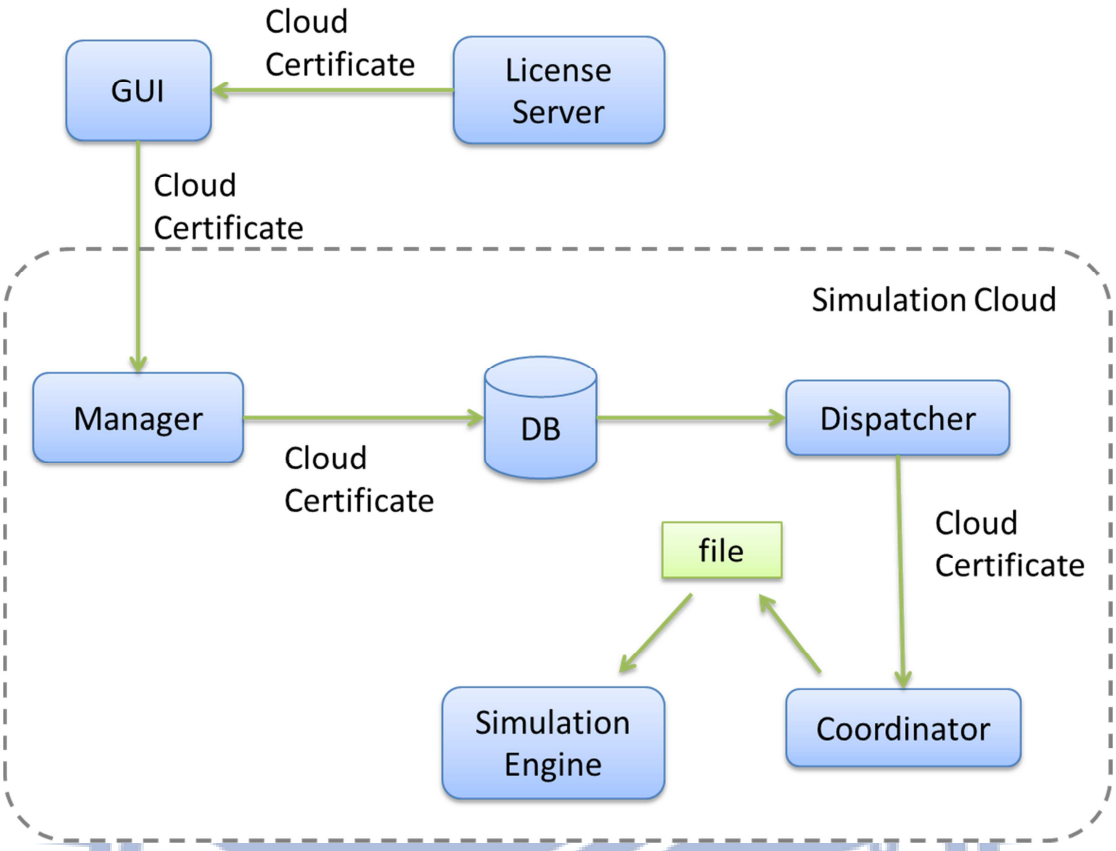

圖 4-9、雲端憑證於各軟體元件的傳遞

第5章 模擬效能測量

設計與實作此雲端系統後，我們希望了解此雲端系統的效能。由於使用者使用此雲端系統時在乎的是執行模擬工作的時間長度，因此我們以測量執行模擬的時間作為效能指標，觀察這個網路模擬雲端系統的效能。

5.1 效能指標與測試環境

本節將說明我們使用的效能指標及效能指標的取得方式，另外也說明我們的測試環境。

處理模擬工作時，Dispatcher 會對每個模擬子工作記錄三個時間點：

- 傳送時間（Submit Time）：Dispatcher 產生此子工作的時間點。
- 開始時間（Start Time）：Dispatcher 分配此子工作給 Coordinator 的時間點。
- 完成時間（End Time）：Coordinator 通知 Dispatcher 子工作完成的時間點。

Dispatcher 分配子工作後到模擬引擎真正開始模擬還有一小段時間，因為中間仍有 IPC 及 fork 子程序等動作，所以 Dispatcher 記錄的開始時間並非模擬引擎真正開始模擬的時間。而完成時間則是 Coordinator 通知 Dispatcher 子工作時，但因為此通知是在模擬引擎結束執行後才送出，因此模擬真正完成的時間會比 Dispatcher 記錄完成時間早一點。但相較於模擬所需要的時間，IPC 及 fork 等動作花的時間來說過小，因此可忽略其影響。我們以「完成時間減開始時間」作為「執行一個模擬子工作所需要的時間」。

在效能測量案例中，對一個模擬工作我們用兩種時間長度作為效能指標：

- 子工作平均執行時間
  如前所述，每個模擬子工作會有執行所需要的時間，將這些時間取平均即為此效能指標。

- 完成所有子工作的時間

此為「最後跑完的子工作的完成時間減第一個開始執行的子工作的開始時間」。

另外我們也使用 Linux 系統中的 time 指令取得模擬引擎的執行時間。time 指令可取得模擬引擎執行一模擬子工作時所花費的實際時間（real time）、使用者時間（user time）及系統時間（system time）。實際時間為模擬引擎從開始到結束的時間。使用者時間為模擬引擎於使用者空間（user space）執行時所花的時間，亦即 CPU 執行模擬引擎的程式碼所花的時間。系統時間為模擬引擎在核心空間（kernel space）所花的時間，亦即 CPU 為執行模擬引擎的系統呼叫（system call）所花的時間，此時間包含在核心中處理 IO 所花的時間。

如系統架構中所述，在我們建置的雲端系統中有五個 Node，作為開啟虛擬機器的實體運算資源，表 5-1 為這些實體機器的硬體配備：

Node	CPU	記憶體
Node1	i7-3770 3.40Ghz（4 實體核心、hyper-threading 為 8 核心）	24 GB
Node2	i7-3770 3.40Ghz（4 實體核心、hyper-threading 為 8 核心）	32 GB
Node3	i7-3770 3.40Ghz（4 實體核心、hyper-threading 為 8 核心）	32 GB
Node4	AMD Opteron™ Processor 6168	48 GB
Node5	i7-2600 3.40GHz（4 實體核心、hyper-threading 為 8 核心）	16 GB

表 5-1、測試環境

我們在 Node 中使用的硬碟為 SSD 硬碟。Node 間網路頻寬為 1Gbps。

此效能測試是測試雲端系統執行模擬的效能，因此用於測試的網路拓撲僅以簡單的有線網路進行 TCP 傳輸。由於模擬執行的速度會因網路狀況、封包數等因素而有不同，因此我們以改變亂數種子產生模擬子工作，降低子工作間因拓撲設定不同而有的時間差距。我們的網路拓撲設定讓一模擬子工作至少需接近一分鐘的執行時間，避免執行時間過短使模擬工作可能在虛擬機器尚未全部開啟即完成。在以下測量案例中，同一個案例中使用的網路拓撲相同。

我們使用虛擬機器執行模擬，在 Openstack 中可指定虛擬機器的運算資源。在以下效能測量案例中，我們的虛擬機器運算資源設置如下：

- CPU 個數：1
- 記憶體：1GB

5.2 效能測量結果

EstiNet 網路模擬器進行模擬時，可一邊模擬一邊將所有封包的傳送及接收記錄至一檔案，稱為 ptr 檔（packet transfer trace file）。使用者可開啟或關閉此記錄功能。在關閉此功能的狀況下，因為模擬引擎會進行許多模擬計算，較屬於 CPU-bound 程序。在開啟此記錄功能的狀況下，模擬期間模擬引擎則 CPU 及 IO 皆有。在接下來的測量案例中，我們會觀察開啟及關閉 ptr 時的效能差異。

5.2.1 測量案例一：寫入 NFS 與本機硬碟之差異

在我們的設計中是將結果檔寫入 NFS 中，透過此測量案例我們希望觀察此設計與將模擬結果檔寫入 Node 本機硬碟在效能上的差異。

我們以 time 指令測量模擬引擎執行的總時間。觀察將結果檔寫入 NFS 以及寫入 Node 本機硬碟時，在開啟、關閉 ptr 時模擬引擎執行總時間的差異。

	寫入 NFS	**寫入 Node 本機硬碟**
開啟 ptr (sec)	175.13	252.7
關閉 ptr (sec)	51.22	48.86
開啟 ptr / 關閉 ptr	**3.41**	**5.17**

表 5-2、ptr 寫入 NFS 與 Node 本機硬碟之執行時間結果一（修改前）

由表 5-2 可看到，我們系統設計將結果檔寫入至 NFS 在開啟 ptr 時增加的時間少於將結果檔寫入 Node 本機硬碟。此結果表示我們將模擬結果檔儲存於 NFS 是一較有效率的設計。

5.2.2 測量案例二：變化虛擬機器數量

在這個測量案例中，我們觀察虛擬機器數量對模擬時間的影響。

5.2.2.1 1 到 10 台虛擬機器

系統參數如下：

- 虛擬機器數量：1 到 10
- 模擬子工作數量：10
- 模擬結果檔：關閉 ptr
- 用於開啟虛擬機器的 Node：Node1、2、3、5，共 4 台，共 16 個實體核心。

因為虛擬機器總數不多，虛擬機器沒有很平均分配在 4 個 Node 上，但這 4 個 Node 的運算能力接近，應無太大影響。

效能測量結果如下：

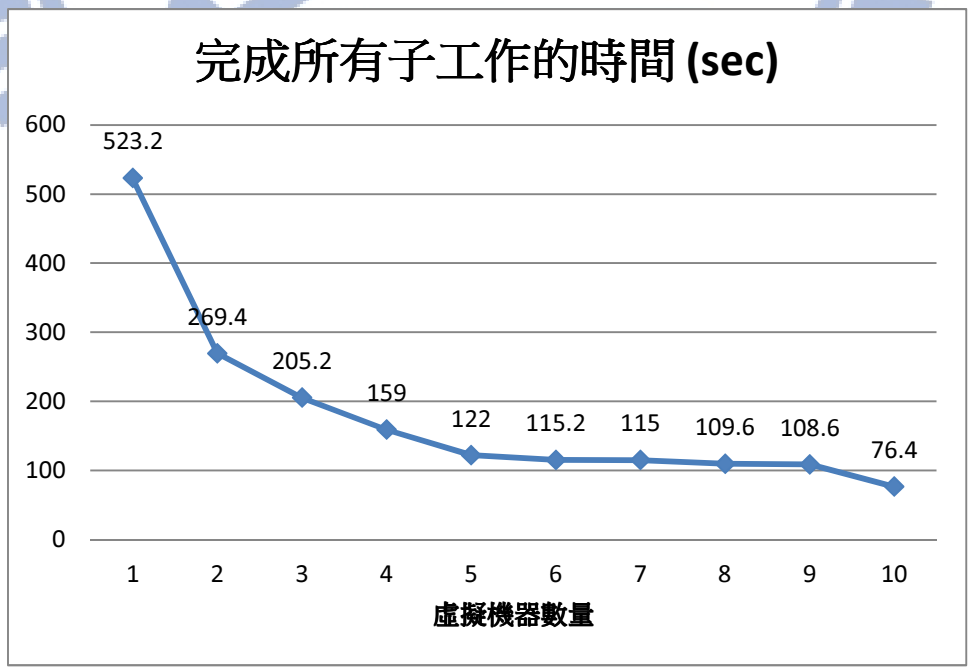

圖 5-1、效能測量案例二之一：完成所有模擬子工作的時間

由圖 5-1 可以看到，虛擬機器數量越多，完成所有模擬子工作所需的時間就越少，這符合同時有多台虛擬機器可提升整體效能、縮短模擬時間的預期。

同時有 2 台虛擬機器執行模擬子工作時，完成所有子工作的時間是只有 1 台虛擬機器時的一半。3 台虛擬機器的時間為 1 台虛擬機器時的 0.39 倍，約為 1/3。因此，在 2 台及 3 台虛擬機器時，模擬時間會減少為原本的 1/2 及 1/3。但到 4 台虛擬機器之後，模擬時間與虛擬機器數量就不再呈 1/N 的關係。

5 到 9 台虛擬機器時，模擬時間減少的幅度極小，10 台虛擬機器時又有明顯的下降。這是因為這個案例有 10 個子工作，在 10 台虛擬機器的情況下，所有虛擬機器只需跑一次模擬即完成所有子工作。5 到 9 台虛擬機器時仍有些虛擬機器需要跑第二輪模擬、會花比較多時間。

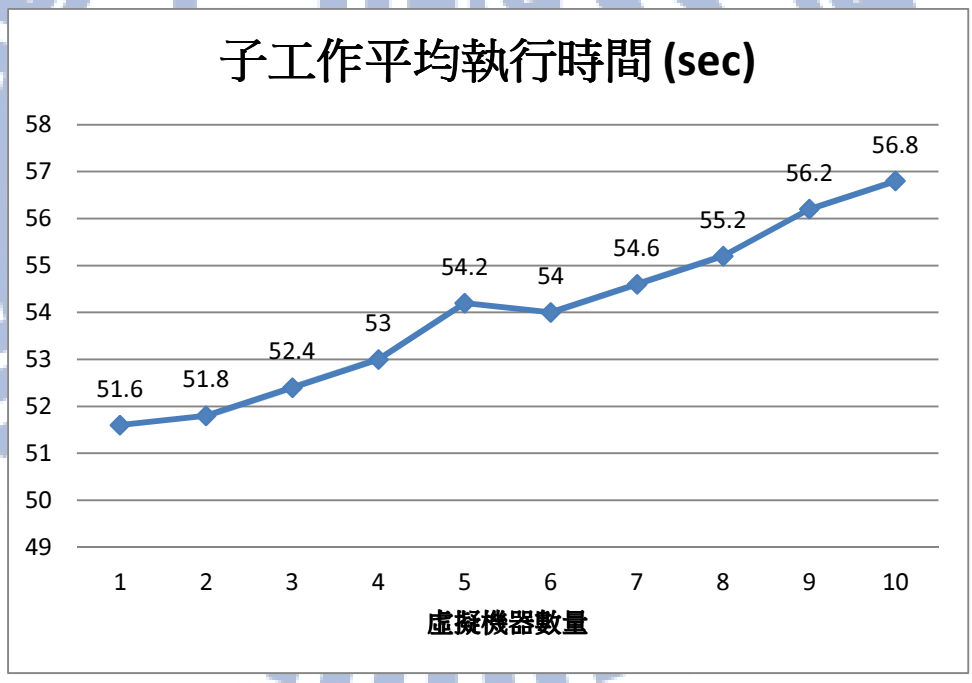

圖 5-2、效能測量案例二之一：子工作平均執行時間

由圖 5-2 可以看到，同時虛擬機器數量越多，執行單一個模擬子工作所需要的時間就越多。在此測量案例中共有 16 個實體核心，分給最多 10 台虛擬機器仍然足夠，因此子工作平均執行時間雖有增加，但增加幅度不大，如表 5-3 所示。

VM 數	子工作模擬時間增加百分比
2	0.4%
3	1.6%
4	2.7%
5	5.0%
6	4.7%
7	5.8%
8	7.0%
9	8.9%
10	10.1%

表 5-3、效能測量案例二之一：子工作模擬時間增加百分比

我們同時計算各子工作間模擬時間的標準差，由表 5-4 可看到各子工作的模擬時間差異不大，表示各模擬子工作所需的時間確實在平均值附近。

VM 數	子工作執行時間標準差 (sec)
1	2.61
2	1.64
3	0.89
4	1.00
5	0.45
6	1.22
7	0.55
8	0.84
9	0.45
10	0.45

表 5-4、效能測量案例二之一：子工作執行時間標準差

5.2.2.2 1 到 20 台虛擬機器

系統參數如下：

- 虛擬機器數量：1 到 20
- 模擬子工作數量：20
- 模擬結果檔：關閉 ptr
- 用於開啟虛擬機器的 Node：Node2 及 3，共 2 台，共 8 個實體核心。

效能測量結果如下：

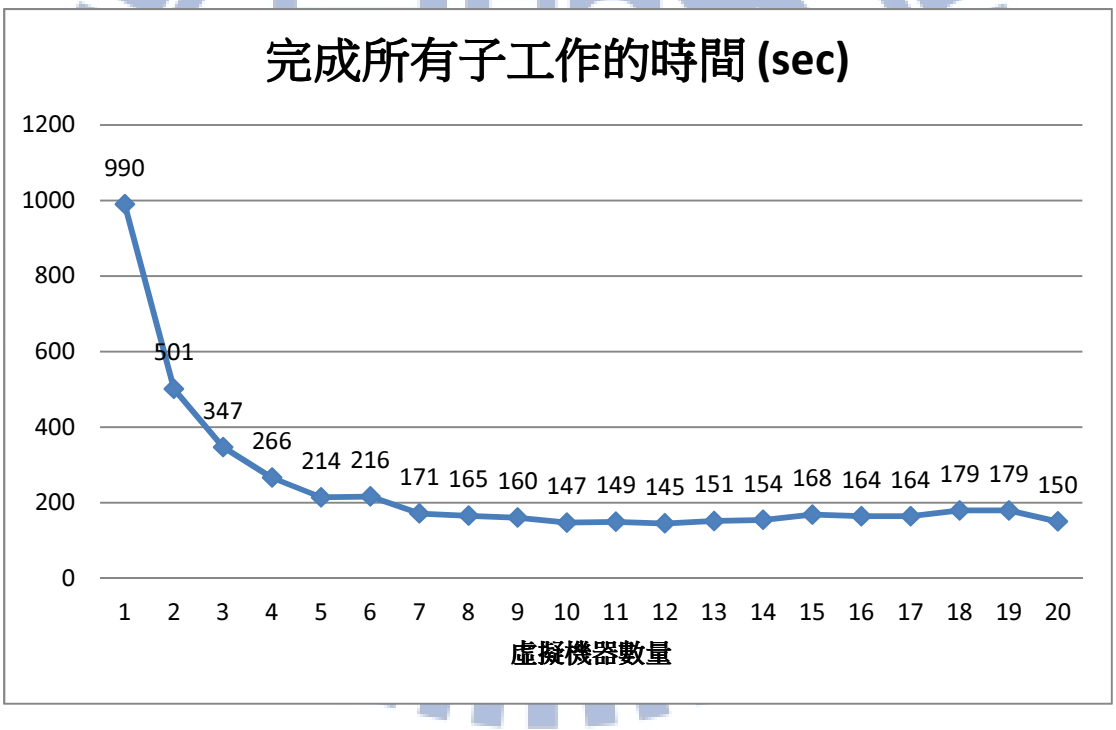

圖 5-3、效能測量案例二之二：完成所有模擬子工作的時間

由圖 5-3 可以看到，虛擬機器數量越多，完成所有模擬子工作所需的時間就越少，這符合同時有多台虛擬機器可提升整體效能、縮短模擬時間的預期。

在 2 至 5 台虛擬機器時，模擬時間與虛擬機器數量約呈反比。6 到 20 台虛擬機器時，模擬時間改變的幅度不大，且 12 台後有稍微增加的情形。這應是當

Node 上有越來越多虛擬機器時，各虛擬機器的效能下降，使最後完成的子工作的完成時間點較容易有變動。

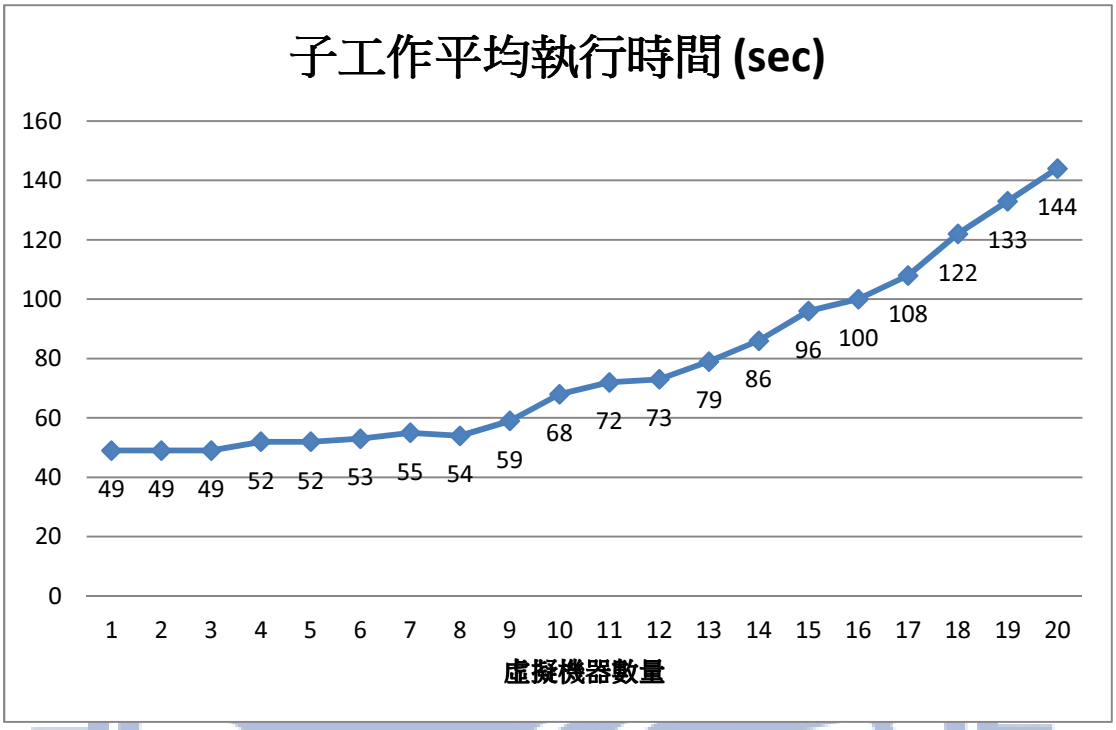

圖 5-4、效能測量案例二之二：子工作平均執行時間

由圖 5-4 可以看到，同時虛擬機器數量越多，執行單一個模擬子工作所需要的時間就越多，符合預期。由圖 5-4 及表 5-5 我們可以看到，模擬時間大致分為三個階段：1 到 8 台、9 到 16 台、17 台以上。1 到 8 台時，增加幅度極小，線段幾乎為水平。9 到 16 台時則有較明顯的增加。17 台以上時則有更明顯的增加幅度。

由於這兩台 Node 分別有四個實體核心，透過超執行緒（Hyper-threading）分別有八個邏輯核心。虛擬機器在 8 台以內時可由實體核心提供運算資源。超過 8 台虛擬機器時則使用到超執行緒所提供的邏輯核心，相較只使用實體核心時有一些 overhead。17 台以上時，則可能會有多台虛擬機器共用一邏輯核心的情形，此會造成 overhead 更為加大。

VM 數	子工作模擬時間增加百分比
2	0.0%
3	0.0%
4	6.1%
5	6.1%
6	8.2%
7	12.2%
8	10.2%
9	20.4%
10	38.8%
11	46.9%
12	49.0%
13	61.2%
14	75.5%
15	95.9%
16	104.1%
17	120.4%
18	149.0%
19	171.4%
20	193.9%

表 5-5、效能測量案例二之二：子工作模擬時間增加百分比

表 5-6 為子工作執行時間的標準差。在虛擬機器數為 1 到 10 台以及 20 台時，標準差較少，表示子工作執行時間沒有太大差距。而 11 到 19 台時，標準差較大，表示當一個 Node 開啟約五個虛擬機器時，有些虛擬機器執行得較快、有些較慢。此是由於一個 Node 開啟五台虛擬機器時，使用到邏輯核心所造成的影響。

VM 數	子工作執行時間標準差 (sec)
1	1
2	1
3	0
4	1
5	1
6	1
7	2
8	1
9	6
10	5
11	12
12	14
13	14
14	20
15	18
16	24
17	25
18	23
19	20
20	3

表 5-6、效能測量案例二之二：子工作執行時間標準差

5.2.2.3 1 到 50 台虛擬機器

系統參數如下：

- 虛擬機器數量：1、2、3、4、5、6、7、8、9、10、11、15、20、25、30、35、40、45、50
- 模擬子工作數量：100
- 模擬結果檔：關閉 ptr
- 用於開啟虛擬機器的 Node：Node1 至 Node5，共 5 台

效能測量結果如下：

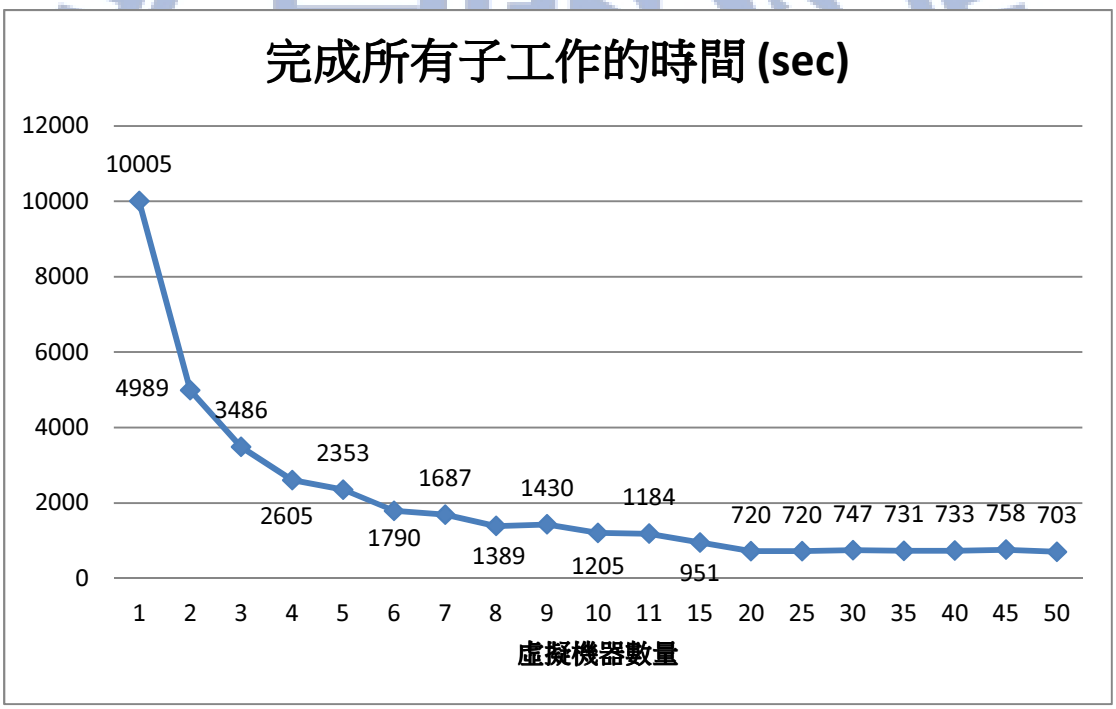

圖 5-5、效能測量案例二之三：完成所有模擬子工作的時間

由圖 5-5 可以看到，虛擬機器數量越多，完成所有模擬子工作所需的時間就越少，這符合同時有多台虛擬機器可提升整體效能、縮短模擬時間的預期。

在 2 至 8 台虛擬機器時，模擬時間與虛擬機器數量約呈反比。9 到 20 台虛擬機器時，模擬時間減少的幅度減小。20 到 50 台時，模擬時間改變的幅度不大，

且其中有稍微增加的情形。這應是當 Node 上有越來越多虛擬機器時，各虛擬機器的效能下降，使最後完成的子工作的完成時間點較容易有變動。

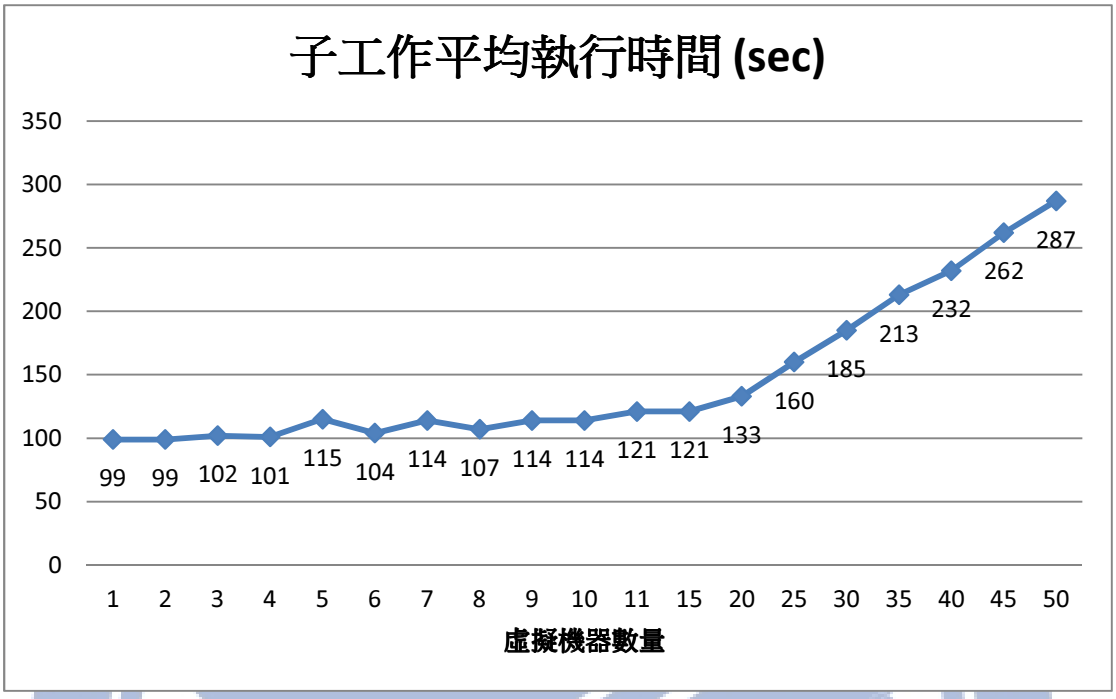

圖 5-6、效能測量案例二之三：子工作平均執行時間

由圖 5-6 可以看到，同時虛擬機器數量越多，執行單一個模擬子工作所需要的時間就越多，這符合我們的預期。由圖 5-6 及表 5-7 我們可以看到，當有 20 台以上的虛擬機器時，子工作所需模擬時間的增加幅度變得比 20 台以下的虛擬機器大許多。

VM 數	子工作模擬時間增加百分比
2	0.0%
3	3.0%
4	2.0%
5	16.2%
6	5.1%
7	15.2%
8	8.1%
9	15.2%
10	15.2%
11	22.2%
15	22.2%
20	34.3%
25	61.6%
30	86.9%
35	115.2%
40	134.3%
45	164.6%
50	189.9%

表 5-7、效能測量案例二之三：子工作模擬時間增加百分比

綜合以上三種虛擬機器數量的效能測量結果。我們可以看到完成所有模擬子工作所需的時間虛擬機器由 1 台往上增加時，會先有大幅度降低。在虛擬機器數量約為模擬子工作數量的一半時，模擬時間不再有明顯的降低。當虛擬機器對實體資源的競爭較為激烈時會有些許的上升。這個結果表示，在此系統組態下，當虛擬機器數量超過模擬子工作數量的一半時，模擬所需時間便難以再有相當的提升。因此，使用者以模擬子工作數量一半的虛擬機器執行一模擬工作，可獲得較大的效益。

5.2.3 測量案例三：開啟與關閉 ptr 的效能差異

在這個測量案例中，我們觀察開啟 ptr 記錄檔及關閉 ptr 記錄檔的差異。

5.2.3.1 10 台虛擬機器

系統參數如下：

- 虛擬機器數量：1 到 10
- 模擬子工作數量：10
- 用於開啟虛擬機器的 Node：Node2、3，共 2 台

效能測量結果如下：

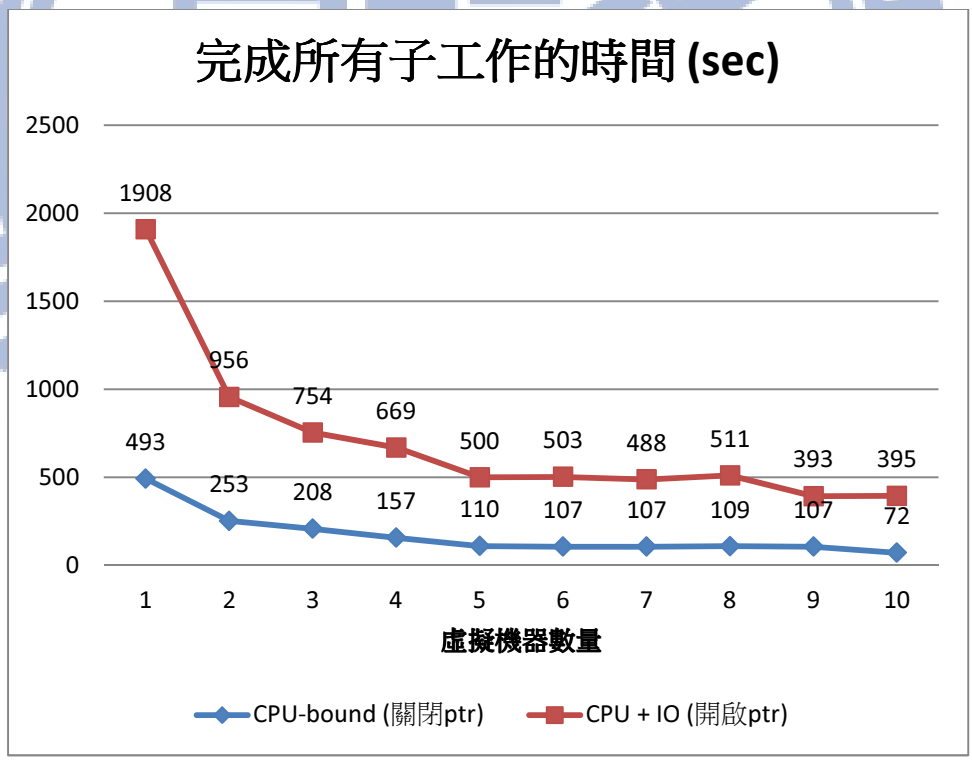

圖 5-7、效能測量案例三之一：完成所有模擬子工作的時間

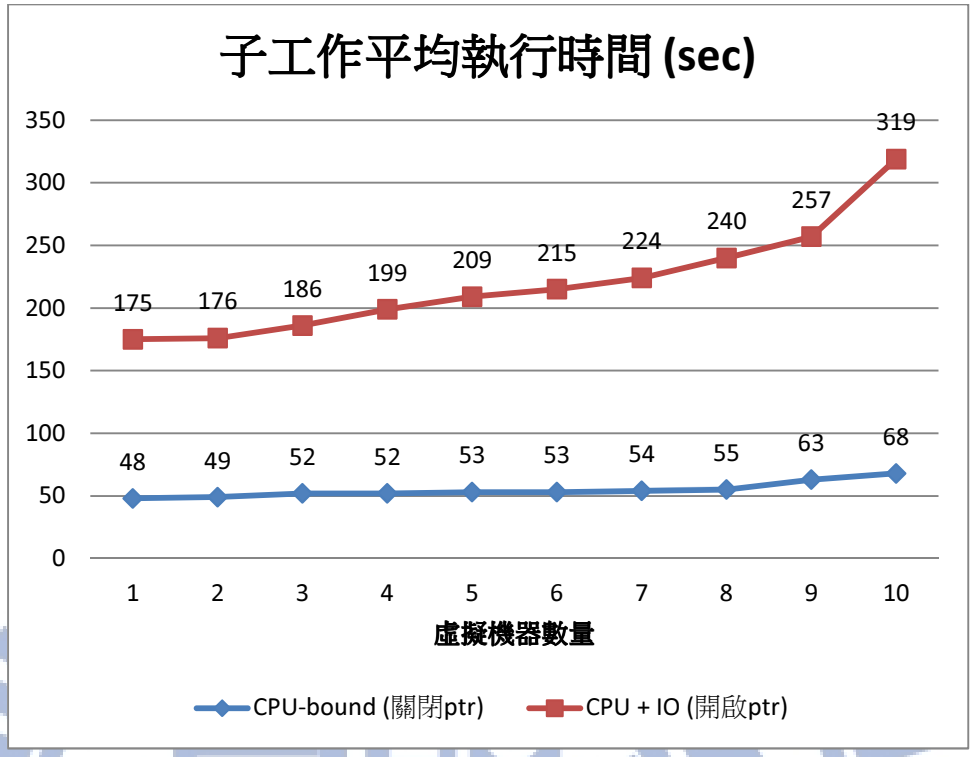

圖 5-8、效能測量案例三之一：子工作平均執行時間

由圖 5-7 跟圖 5-8 可以看到在完成所有子工作時間以及子工作平均執行時間上，開啟 ptr 的時間是關閉 ptr 的 3 至 4 倍。

5.2.3.2 20 台虛擬機器

系統參數如下：

- 虛擬機器數量：1 到 20
- 模擬子工作數量：20
- 用於開啟虛擬機器的 Node：Node2、3，共 2 台

效能測量結果如下：

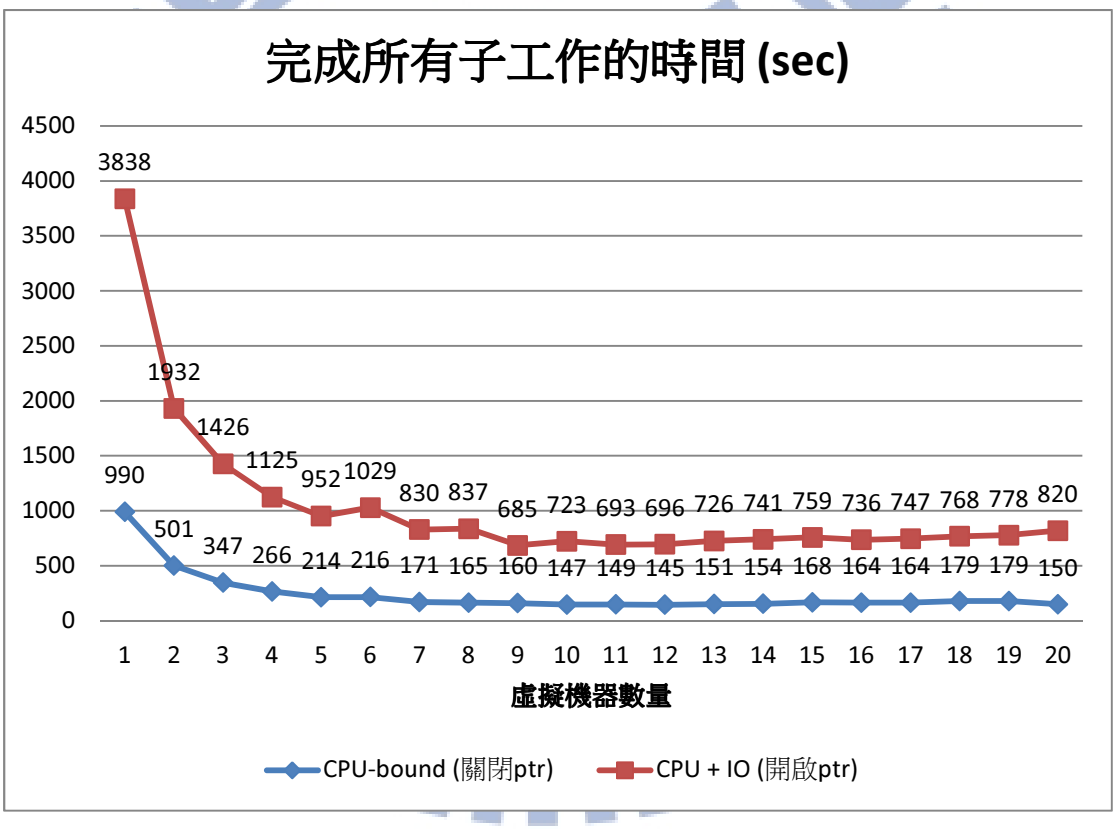

圖 5-9、效能測量案例三之二：完成所有模擬子工作的時間

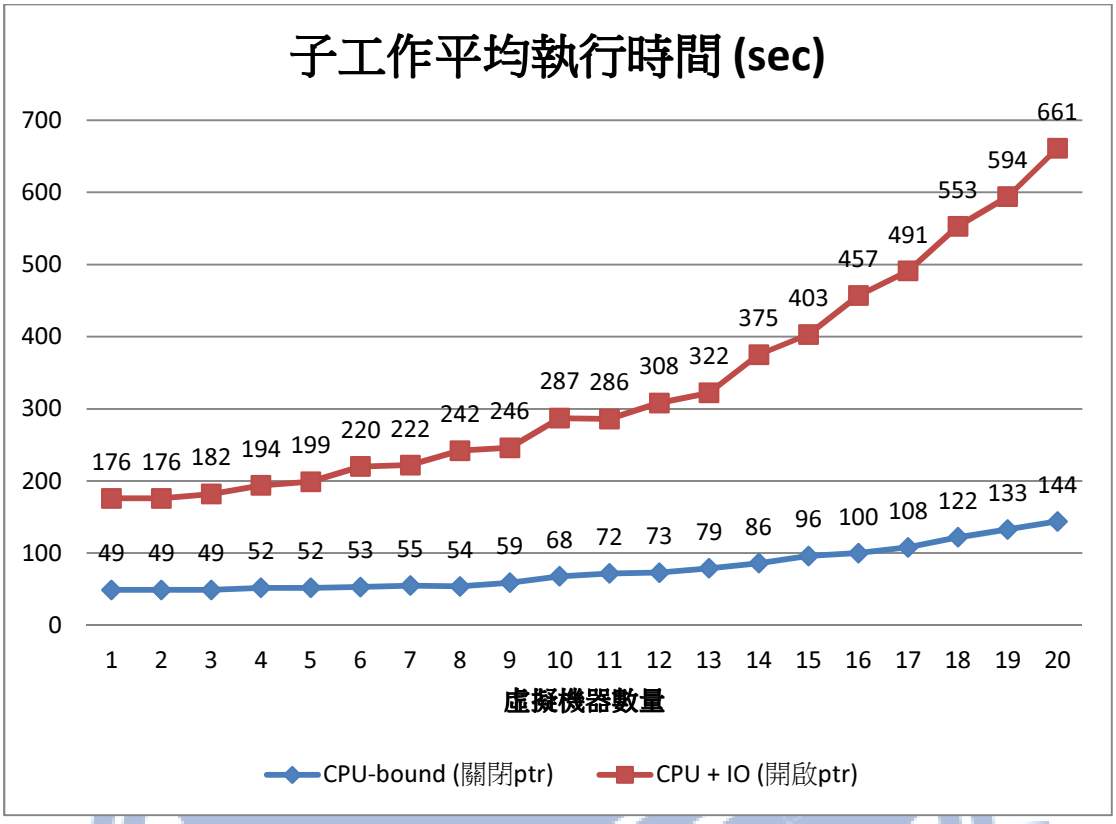

圖 5-10、效能測量案例三之一：子工作平均執行時間

在虛擬機器數量及模擬子工作增加時，由圖 5-9 跟圖 5-10 同樣可觀察到在完成所有子工作時間以及子工作平均執行時間上，開啟 ptr 的時間是關閉 ptr 的 3 至 5 倍。

5.3 效能結果分析與驗證

從效能測量結果可以看到，開啟 ptr 及關閉 ptr 的模擬時間有 3 到 5 倍的差距，我們希望了解是什麼因素造成這個差距。直覺上，由於開啟 ptr 與關閉 ptr 的差別是模擬引擎是否有寫入檔案，因此猜測此差距可能是 I/O 造成的。以下將說明分析此差距的原因及驗證。

表 5-8 以 1 台虛擬機器執行模擬子工作得到的數據。由此數據可觀察到開啟 ptr 時使用者時間增加佔總增加時間的比例為 42.2%、系統時間增加佔的比例為 57.8%，系統時間增加的比例稍多，但與預期上系統時間的增加會遠大於使用者時間不同。此結果表示開啟 ptr 時，除了寫入 ptr 檔的 I/O 操作會增加模擬時間外，CPU 的運算時間也會增加。

	使用者時間 (sec)	**系統時間 (sec)**	**加總 (sec)**
開啟 ptr	73.8	86.9	160.7
關閉 ptr	23.7	18.49	42.19
開啟 ptr – 關閉 ptr	**50.1 (42.2 %)**	**68.41 (57.8 %)**	**118.51 (100 %)**

表 5-8、模擬引擎開啟、關閉 ptr 的使用者時間及系統時間結果一（修改前）

我們希望由模擬引擎的程式碼驗證開啟 ptr 時確實會增加 IO 操作及 CPU 計算。以下說明開啟 ptr 檔時模擬引擎的相關實作。

EstiNet 網路模擬器以 event-driven 的方式進行模擬。在模擬引擎中有一虛擬時間，表示模擬世界目前進展的時間。各種封包傳送、接收等動作會被當作一個事件（event），並插入（insert）至事件堆積（event heap）。每個事件會有 expired time，expired time 比目前虛擬時間小的事件會被模擬引擎處理。在模擬引擎中，推進虛擬時間、取出事件並加以處理的部分稱為排程器（Scheduler）。

「寫入 ptr 檔」在模擬引擎中被視為一個事件、由排程器（Scheduler）處理。排程器在虛擬時間中每 100 ms 會處理一次 ptr 事件。排程器處理一個 ptr 事件時會處理所有 expired time 比目前虛擬時間小的 log 事件。

協定模組傳送與接收封包時將相關的 log 事件插入（insert）至 log 佇列（queue）。ptr 檔中一筆封包傳送或接收的記錄需要一對 log 事件。傳送記錄（TX record）需要 StartTX 跟 SuccessTX/DropTX log 事件。接收記錄（RX record）需要 StartRX 跟 SuccessRX/DropRX log 事件。

實作上，協定模組初始化時以附加（append）模式開啟 ptr 檔，並插入第一個 ptr 事件到排程器的事件堆積（event heap），此事件的 expired time 為 100 ms 後。協定模組在傳送及接收封包時會呼叫一些函式取得此封包傳輸的相關資訊，以這些資訊產生 log 事件，最後將 log 事件插入至 log 佇列。

排程器處理一個 ptr 事件時，會處理所有 expired time 比目前虛擬時間小的 log 事件，處理方式主要為找 log 的配對及寫入 ptr 檔，流程如下：

1 配對 log 事件。

1.1 log 事件為 StartTX 或 StartRX，則將此事件插入至一串列（list）。

1.2 log 事件為 SuccessTX 或 DropTX（SuccessRX 或 DropRX 同理），則以迴圈搜尋上述串列中與此事件對應的 StartTX（StartRX）事件，找到後記錄此配對。

2 用迴圈在上述串列中搜尋配對好的一對 log 事件、寫入 ptr。

ptr 檔一筆記錄有多個欄位。寫入 ptr 檔的程式碼是一次寫入一個欄位，即對一個欄位便呼叫一次 fwrite()寫入。在 802.3 協定模組中，寫入一筆記錄會呼叫 14 次 fwrite()。

我們從程式碼中發現每寫入一筆 ptr，模擬引擎會呼叫十多次的 fwrite()，這是造成 I/O 時間增加的原因。CPU 時間的增加則是由於開啟 ptr 時，模擬引擎需多處理 ptr 事件及 log 事件，如取得封包資訊、搜尋配對等。

我們修改模擬引擎寫入 ptr 時呼叫多次 fwrite()部分的程式碼。由於 ptr 檔是二進位檔（binary file），因此先以 memcpy()將要寫入的記錄資料存在一個記憶體空間中，最後再呼叫 fwrite()將記錄寫入 ptr 檔案。

	使用者時間 (sec)	系統時間 (sec)	加總 (sec)
開啟 ptr	60.3	40	100.3
關閉 ptr	23.72	18.1	41.82
開啟 ptr – 關閉 ptr	**36.58 (62.5%)**	**21.9 (37.5%)**	**58.48 (100%)**

表 5-9、模擬引擎開啟、關閉 ptr 的使用者時間及系統時間結果二（修改後）

表 5-9 為修改模擬引擎後再次測量的時間數據，由此表可觀察到修改後、開啟 ptr 時使用者時間增加佔總增加時間的比例為 62.5%、系統時間增加所佔的比例為 37.5%。

	寫入 NFS	寫入 Node 本機硬碟
開啟 ptr (sec)	113.4	134.37
關閉 ptr (sec)	50.6	48.32
開啟 ptr / 關閉 ptr	**2.24**	**2.78**

表 5-10、ptr 寫入 NFS 與 Node 本機硬碟之執行時間結果二（修改後）

表 5-10 是修改模擬引擎後，針對寫入 NFS 與寫入 Node 本機硬碟以 time 指令測量執行實際時間的結果。由表 5-2 及表 5-10 的比較，可以看到在修改後開啟 ptr 的實際執行時間無論是寫入 NFS 還是寫入 Node 本機硬碟都有減少，如表 5-11 的比較。

	寫入 NFS		寫入 Node 本機硬碟	
	修改前	修改後	修改前	修改後
開啟 ptr (sec)	175.13	113.4	252.7	134.37
關閉 ptr (sec)	51.22	50.6	48.86	48.32

表 5-11、ptr 寫入 NFS 與 Node 本機硬碟執行時間修改前後比較

這代表我們的修改確實降低系統時間，並且讓系統時間增加佔總增加時間的比例下降。此表示過多 fwrite()確實是造成系統時間增加的因素。

第6章 未來工作

本論文所設計與實作的網路模擬雲端系統已可運行併行模擬之功能，也對模擬引擎異常結束的部分進行相關處理。但此雲端系統中仍有許多可繼續研究、開發的議題，在此提出以下幾點。

- 預先開啟虛擬機器

在本論文系統的實作中，Manager 雖能判斷系統中是否已有開啟的虛擬機器並加以分配，但我們尚未實作預先開啟虛擬機器的功能，因此在虛擬機器的自動啟動上皆為收到模擬工作後才開啟新的虛擬機器。由於開啟虛擬機器需要一段時間，若能以預先開啟虛擬機器提供服務，可減少此系統處理一模擬工作所需的時間。

- 自訂模擬引擎與應用程式之安全性

此系統提供使用者可執行自行開發的模擬引擎及應用程式，在本論文的實作中僅以 NFS 檔案權限避免惡意使用者存取他人的模擬結果檔，並以封閉網路的方式避免虛擬機器成為攻擊外界網路的跳板。但目前仍無法避免惡意使用者利用虛擬機器攻擊雲端系統中其他虛擬機器或主機，亦無相關偵測機制。

- 系統容錯（Fault tolerance）與錯誤恢復（Error Recovery）

對一雲端系統來說，系統容錯與錯誤恢復是相當重要的。在本論文的設計與實作中將 Manager 的執行狀態儲存於資料庫，但並沒有進一步實作系統容錯與錯誤恢復的相關機制。

第7章 結論

在前面的章節說明了我們如何以 EstiNet 網路模擬器及 Openstack 平台設計與實作併行模擬、使用者自訂模擬引擎及應用程式兩大主要功能。此系統以此兩大功能讓使用者可節省進行模擬實驗所需的時間，並且讓模組開發人員可使用雲端系統的運算資源並以自行開發的模組進行模擬。

為了讓前述兩大功能得以順利進行，我們同時設計並實作儲存空間管理以及認證機制。儲存空間管理用於管控使用者於此系統中的檔案。認證機制則整合 EstiNet 網路模擬器既有之認證機制，使雲端系統中的模擬引擎可以使用者購買的模擬能力進行模擬。

除了讓模擬可以正常運作外，為因應使用者自行開發模擬引擎時可能發生的錯誤狀況，我們也對模擬引擎異常結束的狀況進行相關處理，使虛擬機器可再次執行模擬引擎並且令使用者可取得異常結束的相關訊息。

我們對此雲端系統進行效能上的測量及分析，結果顯示多台虛擬機器同時進行模擬可減少模擬工作所需的時間。由不同數量虛擬機器進行模擬所得的效能結果，我們發現在我們的實驗環境中，使用者以模擬子工作數量一半的虛擬機器執行一模擬工作，可獲得較大的效益。在開啟與關閉 ptr 的比較中，我們發現在同樣使用虛擬機器進行模擬的狀況下，虛擬機器由 NFS 存取相關檔案所增加的模擬時間比由實體機器的本機硬碟存取相關檔案來得少，代表以 NFS 作為儲存空間的設計是較有效率的。最後，我們也針對開啟與關閉 ptr 的效能差異進行分析，找出開啟 ptr 時造成模擬時間增加的原因。

此網路模擬雲端系統已可提供網路協定研究者進行併行模擬，節省網路模擬實驗所需的大量時間及人力。未來若能在安全性、系統容錯、效能等議題上有更進一步的研發與整合，將能使此系統更臻完善。

參考文獻

[1] EstiNet network simulator and emulator, available at http://www.estinet.com

[2] Openstack, available at http://www.openstack.org

[3] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin, “The Design and Implementation of the NCTUns 1.0 Network Simulator”, Computer Networks, Vol. 42, Issue 2, June 2003, pp. 175-197. (SCI)

[4] S.Y. Wang, “NCTUns 1.0”, In the column "Software Tools for Networking", IEEE Network, Vol.17, No.4, July 2003. (SCI)

[5] C.L. Chou, “A Network and Traffic Simulator for Wireless Vehicular Communication Network Research”, Ph.D thesis, National Chao Tung University, Shinchu, Taiwan, 2009.

[6] S.Y. Wang, C.L. Chou, C.C. Lin, “The Design and Implementation of the NCTUns Network Simulation Engine”, Simulation Modelling Practice and Theory, 15 (2007) 57-81. (SCI)

[7] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas, "Cloud Computing Synopsis and Recommendations", NIST Special Publication 800-146, May 2012

[8] Peter Mell, and Timothy Grance, "The NIST Definition of Cloud Computing", National Institute of Standards and Technology Special Publication 800-145, September 2011

[9] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I., "A View of Cloud Computing", Communications of the ACM, vol 53, no. 4, p. 50-58, 2010