國立中山大學資訊工程學系

碩士論文

Department of Science and Engineering

National Sun Yat-sen University

Master Thesis

基於虛擬系統之 Watchdog 設計與實作

Design and Implementation of Watchdog in the Virtual System

研究生：鄭又新

Yu-Hshin Cheng

指導教授：王友群 博士

You-Chiun Wang, Ph.D.

中華民國 103 年 7 月

July 2014

國立中山大學資訊工程學系

碩士論文

Department of Science and Engineering

National Sun Yat-sen University

Master Thesis

基於虛擬系統之 Watchdog 設計與實作

Design and Implementation of Watchdog in the Virtual System

研究生：鄭又新

Yu-Hshin Cheng

指導教授：王友群 博士

You-Chiun Wang, Ph.D.

中華民國 103 年 7 月

July 2014

國立中山大學研究生學位論文審定書

本校資訊工程學系碩士班

研究生鄭又新（學號：M013040031）所提論文

基於虛擬系統之Watchdog設計與實作
Design and Implementation of Watchdog in the Virtual System

於中華民國 103 年 7 月 18 日經本委員會審查並舉行口試，符合碩士學位論文標準。

學位考試委員簽章：

召集人 周承復 周承復　　委　員 王友群 王友群

委　員 謝文雄 謝文雄　　委　員 陳璽煌 陳璽煌

委　員 劉建興 劉建興　　委　員 _____

指導教授(王友群) 王友群 (簽名)

誌謝

首先我要感謝我的家人，如果不是他們在我求學的路上一路陪伴我，我不可能會有機會來到中山大學就讀，更不可能一路讀上碩士班然後順利畢業，在此以本論文獻給我最摯愛的家人。

再來則是我在讀書階段教過我的所有老師們，感謝你們教導我們在這條路上需要的基本知識，並且為了我們能不繞遠路，細心教導學業上的重點；在口試時，各位口委老師們的教導、建議讓我能將研究的題目能再往上推進一大步，真心謝謝你們；而不管是大學時代的導師，或者到研究所的指導教授，除了課業上的教授外，另外又扮演著照顧、關懷我們的角色，辛苦你們了。

然後要感謝的是和我一起度過研究室日子的這些同學們，雖然各位在研究的事物都不盡相同，但是如果遇到能幫上忙的部分，大家還是會全力相挺；大家在這段時間中，一起歡樂、學習、搞笑，無論如何你們都是最好的同伴。謝謝有朝勳、冠賢、政修、世宏、銘傳、順昶，你們的陪伴；也感謝淳楷、子華、宗霖、金翼、順賢、俊瑋，各位學弟們加入了這個新環境。

最後在這邊祝福：不管是同屆，或者新一屆，又或是已經在社會上工作的各位，願你們在接下來的路途上即使遇到問題，也能迎刃而解，在這邊獻上感謝與祝福。

摘要

由於現在雲端系統的發展，無論是各大廠商或是個人都可以簡單的向相關單位租用他們的設備來達到他們的需求，例如熟知的各個國外大廠：亞馬遜、Google、微軟、IBM 都已經在這裡有著自己的一席地位。

Docker[1]是個強大的虛擬機器配置系統，使用者可以依照使用需求，透過簡單的指令去配置虛擬機器－Container；讓客戶想要執行的程式放在底下作為一個 Turnkey[2]，然後再讓我們能以 Turnkey 為單位統一管理這些虛擬裝置。

如果這些運行的系統發生故障，想必會造成很大的問題，因此也需要能夠監視這套 Docker 系統（包含 Manager、Ingress、Proxy、Real host、Containers）的 Watchdog，以便於在發生狀況時能及時搶救，免於更大的損失。在這套 Watchdog 中，我們以 ZeroMQ[3]作為連線的工具，和一般 UNIX Domain Socket[4]相比 ZeroMQ 已經模擬了現實生活中許多的連線狀況，並且能處理大部分的問題。

為了方便使用者能在設計服務時使用，本論文會將此 Watchdog 做成 API 的形式來呈現。

關鍵字：看門狗、雲端系統、虛擬機器、入口網站、代理伺服器、實體機器

Abstract

As cloud system development recently, everyone can meet their requirement by renting device resources from companies which provide the services, such as Amazon, Google, Microsoft, IBM, and so on.

Docker is a powerful virtual machine management system for developers and sysadmins to build and run services on containers – as known as a turnkey.

It will become a big trouble when these system fail on running some important services, so we will need a watchdog that can supervise our Docker system, including manager, ingress, proxy, real host and containers, to help us recovering system from a huge loss of data or running processes.

My watchdog uses ZeroMQ tools to cross each different networks. ZeroMQ has already handle a lots of network issues for developers when designing network applications.

In this thesis, I will try to make a watchdog that can be used in cross-network environments and implement it as an API for convenience when others want to use in their service application design for reliability issue.

Keyword：Watchdog、Cloud system、Virtual machine、Cross-network、Docker、Ingress、Proxy、Real host、Linux Container、Turnkey、ZeroMQ

目錄

圖次

表次

第一章　序論

1.1　研究動機與目的

基於 Docker 系統上的層層相連關係，除了製作不可或缺之 Watchdog 外，還需要能提供資訊傳遞，以及能透過上層管理其中 Container[5]之方法，無法單純使用位於本機上所提供的 Watchdog 機制幫忙。

進行這次研究前去找了相關的研究資料：Linux watchdog[6]、Linux supervisor[7]，都是在 Linux 系統中常見的監控方式。

Linux watchdog 具有相當強的系統回復力，符合條件時會重開系統，對於重視 Kernel 存活的使用上有著深遠的影響，唯一的問題是使用者需要對他具有相當的熟悉度後才好上手，寫程式時可能發生 ioctl () 沒有支援相關函式，所以操作 Watchdog 時會引發錯誤…等，對於不須探究太深知識之應用上顯得頗難運用。

Linux supervisor 可讓使用者簡單藉由撰寫 Script 達到對指定 Program 監視的效果，重開時是針對該程式執行，並不會直接重開系統，對於只是管理 User-space 的程式相當方便，就管理監控的部分相當值得參考。僅僅透過 Script 的方式就能監控程序，甚至可以指定 UNIX Socket，唯一不足的部分應該是如果發生錯誤時他不會和其他程式溝通，在虛擬系統中需要的不只是確保環境正常，還有發生錯誤時還能通報上層系統的能力這點上 Supervisor 沒辦法獨立完成。

Linux watchdog 和 Supervisor 主要都是以 Commands 的型態操作，在使用者撰寫程式時不容易協助使用，本論文動機在於提供方便的 Watchdog API，使得使用者在虛擬系統中製作服務時，能有 Watchdog 互相連接可確保更上層的系統知道下方的 Watchdog 目前的狀況，藉此架設更加可靠的服務環境。

1.2 相關研究

任何重要系統中若是單就程式本身的例外預防，往往無法完全防止各種狀況發生，因此使用 Watchdog 來輔助系統使之具有更高的可靠度並防止系統本身無法處理的狀況是相當常見的方式。

Watchdog 機制可幫忙簡單記錄 Log、不過主要的功能為確定目標是否依然在活動（Active），若無法探測則需要負責重開系統，以維持系統正常運作。

1.2.1 Linux Watchdog

Linux 系統上具備了的 Hardware Watchdog 和 Software Watchdog[8]，部分硬體設計時就已經支援 Watchdog 的機制，另一部分透過 Kernel 軟體支援成為 Watchdog module 使用者可在安裝 Watchdog Daemon 後使用，無論是哪一種用法上都是先開啟/dev/watchdog 後持續對此裝置寫入，將 watchdog timer 重置，如果未在期限內繼續寫入，或者發生了不符合 configure 中的條件，Watchdog 即會立即將系統重開；最後使用者可以規定的形式關閉 Watchdog，但是如果編譯 Kernel 時將 nowayout 參數開啟（CONFIG_WATCHDOG_NOWAYOUT），Watchdog 就無法被其他方式關閉。

以下為 Soft Watchdog 的一使用範例 改寫至 How to Use Linux Watchdog[9]：

```
1  int fd = open("/dev/watchdog", O_RDWR);
2  if (fd == -1) {
3      fprintf(stderr, "Error: %s\n", strerror(errno));
4      exit(EXIT_FAILURE);
5  }
6  int interval = 5 /* Watchdog timeout interval (in secs) */
```

```
7   if (ioctl(fd, WDIOC_SETTIMEOUT, &interval) != 0) {
8       fprintf(stderr, "Error: Set watchdog interval failed\n");
9       exit(EXIT_FAILURE);
10  }
11  char kick_watchdog;
12  do {
13      kick_watchdog = getchar();
14          switch (kick_watchdog) {
15          case 'w':
16              write(fd, "w", 1);
17              fprintf(stdout, "Kick watchdog through writing over device file\n");
18              break;
19          case 'i':
20              ioctl(fd, WDIOC_KEEPALIVE, NULL);
21              fprintf(stdout, "Kick watchdog through IOCTL\n");
22              break;
23          case 'x':
24              fprintf(stdout, "Goodbye !\n");
25              break;
26          default:
27              fprintf(stdout, "Unknown command\n");
28              break;
29          }
30  } while (kick_watchdog != 'x');
31  write(fd, "V", 1);
```

```
32  /* Closing the watchdog device will deactivate the watchdog. */
33  close(fd);
```

範例中為了保持 Watchdog 運作持續的寫入“w”進入該 dev 中，或者可以藉由 ioctl 直接使用 WDIOC_KEEPALIVE 參數操作 device。最後透過寫入 "V" 表示結束使用 Linux Watchdog。

某些系統運行上具有永續進行的需求，而 Linux watchdog 通常就是為了幫助使用者需要掌握這些系統的狀態、CPU load、Memory leak、甚至是主機運作溫度…等需由電腦輔助才能應付的情形，透過他來提升對風險的處理。

1.2.2 **Linux Supervisor**

Linux Supervisor 只有提供 Unix based System 使用，主要用來做 Processes 間的管理要務，特別是能提供一個固定的管理介面，讓使用者能一致性的管理這些需要長駐的行程。以下介紹一段在 Supervisor 中使用的簡單 Script 範例：

使用者需要填寫.conf 檔並置於/etc/supervisor/conf.d/之下，.conf 中應包含的項目如下[10]：

```
[program:long_script]                    //表示需監控的程式；
command=/usr/local/bin/long.sh           //指出該 program 的實際位置；
autostart=true                           //告知 Supervisor 在啟動時應去執行
autorestart=true                         //若此 program 離開，Supervisor 需做
處理
stderr_logfile=/var/log/long.err.log     //該 program 的 stdout 訊息會記錄在此
stdout_logfile=/var/log/long.out.log     //該 program 的 stderr 訊息會記錄在此
```

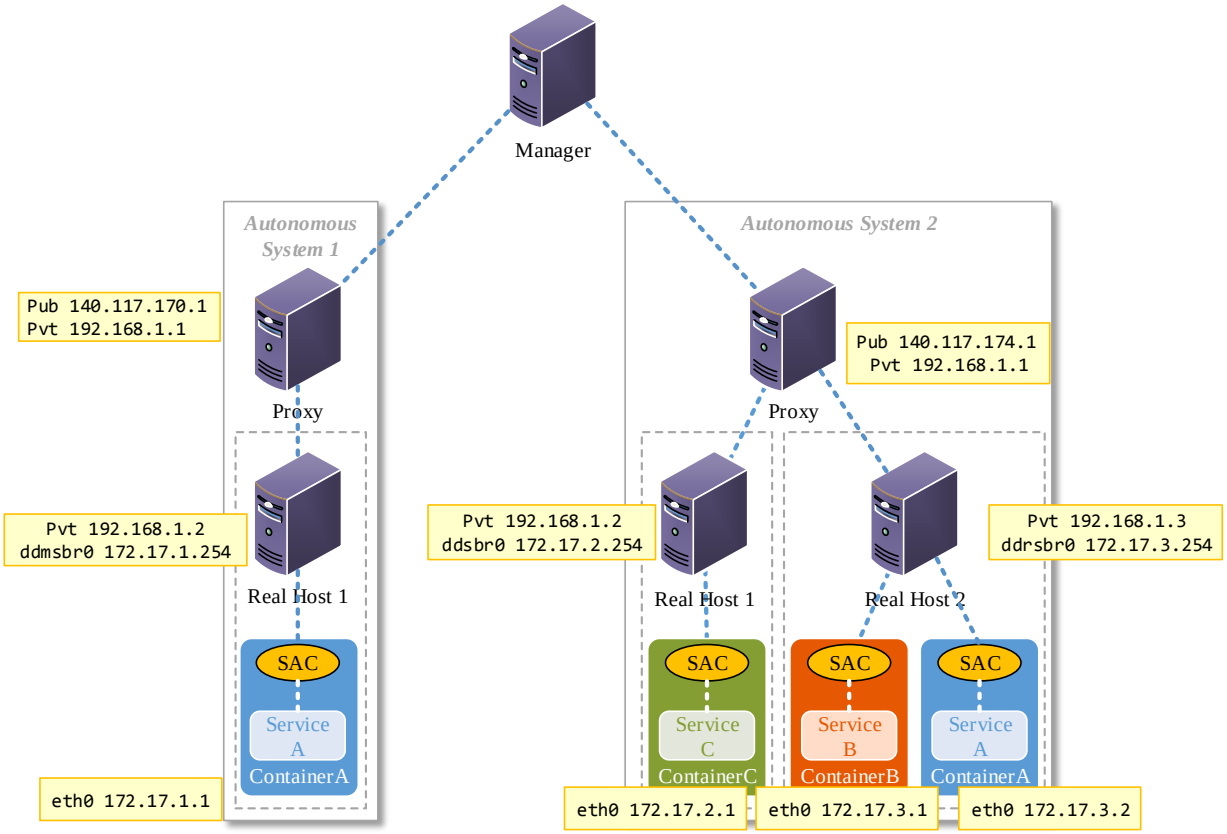
任何對/etc/supervisor/conf.d 裡面的內容增減均需透過 $ supervisorctl reread 和 $ supervisorctl update 這兩個指令來告訴 supervisor 有做任何更改。

為了操作 Supervisor 可以進入 supervisorctl 中查看目前監控下的 Process 狀態，這邊主要分 RUNNING、STOPPED 兩種，注意：若需要 Supervisor 幫忙觀察則 Program 必須是執行於前景，否則無法正確地知道 program 的當前狀態，更可透過 start、restart、stop 這些指令改變目前監控的項目應該表示的狀態。

Supervisor 使用簡單並且能掌握 Process 間的狀況，對於不是需要相當重要的系統性的 Reboot 而只是 User 自行定義的一些程式而言，可說是既方便又實在。

第二章　相關背景知識介紹

2.1　Docker 虛擬系統架構介紹



Docker 系統中具備了一主控 Manager，往下分出許多系統自治區 AS（Autonomous System），每一 AS 中都具備了各自的 Private IP[11]保證和外部分離，各自管理著底下的 Linux Container。

承租 Container 的使用者可於其中製作自己需要的服務，然後透過 Docker 方的入口網站取得該服務內容。

系統中會將服務內容相同的 Container 放到其他 AS 中，確定系統不會搞混底下的內容重覆，如果流量超越區域負荷，系統就能幫忙將超過的部分轉往其他區域中平衡負擔。

2.2 ZeroMQ

ZeroMQ，又可稱為 0MQ, or zmq。ZeroMQ 重新封裝了 Berkeley Socket[13]並可藉由 TCP、ipc、inter-process 或者 multicast ……等協定傳輸訊息，並且能處理在傳輸訊息時訊息間不可分割的問題。

Socket 間允許 N – N Pattern 的連結方式，可用來實現 Pub – Sub、Task 分配、Request – Reply 等…常見的應用法。處理速度之快已能作為叢集系統中之網路設計。

提供非同步 I/O 之 API 可於 multicore 軟體的應用，支援 C、C++、C#、Java、Perl、PHP、Python、Ruby…等二十多種程式語言、跨平台。由 iMatix 公司和大量貢獻者組成的社群共同開發，是通過 LGPLLv3[14]認可之 Open Source。

2.2.1 協定

2.2.1.1 TCP

ZeroMQ 最廣為使用的協定，以可靠、單播為主，在各種應用上都能得到不錯的效果。定址時需填入的 ZeroMQ address 字串格式為：tcp://endpoint，endpoint 的格式需求如下：

使用 zmq_bind () 需指名欲接受的介面以及使用的 port，介面的形式有：

1. *：表示所有可用的網路介面。
2. 以 IPv4 為主的網址。Ex: tcp://140.117.171.225:5555 表示接受從 140.117.171.225 Port 5555 來的連線。
3. 其他由 OS 定好的介面。

使用 zmq_connect () 時必須指定某個特定的對象的位置，格式為：

1. 以 DNS 網域名稱為名的位置
2. 以 IPv4 為主的網址。Ex: tcp://140.117.171.225:5555 連線到 140.117.171.225 Port 5555。

2.2.1.2 IPC

ZeroMQ 將使用系統內建的 ipc 機制進行傳輸，此方式僅限於有提供 UNIX domain socket 的系統中。

定址所需的格式為：ipc://endpoint，其中 endpoint 格式如下：

使用 zmq_bind () ，endpoint 需符合系統格式且唯一的路徑名稱。

使用 zmq_connect ()，endpoint 指定的路徑必須是之前由 zmq_bind () 產生的檔案路徑。

2.2.2 Pattern

2.2.2.1 REQ <-> ROUTER

簡單易使用的 Server-Client 形式，一個 Router Server 可同時應付許多的 Req Client。Router Socket 在這邊具有 Proxy 將訊息自動分給後端處理的特性，可以有效讀取並分發從中經過的各種 Frame。

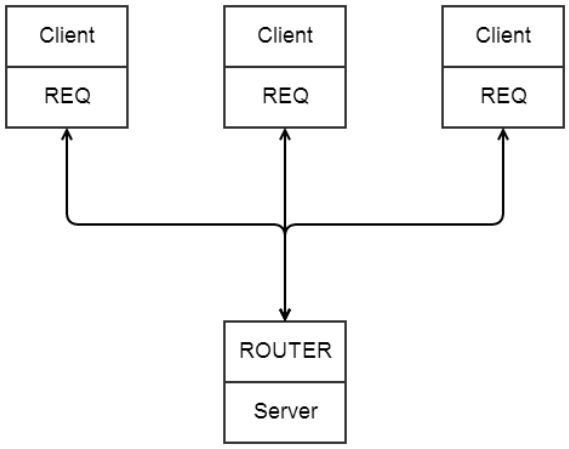


圖2.1 Request and Router Pattern

2.2.2.2 ROUTER <-> ROUTER

進階的 Req – Router 使用方式，由於 Socket 的兩邊都具有不只一份的需求，所以需要這種特別針對多對多的處理方式。可以只接指定須由對方哪個部分接收，兩邊在溝通時均須在一開始標註由誰收封包，否則 Router Socket 將一律過濾掉。

2.2.3 訊息傳遞

訊息傳遞的部分借用另一套由 ZeroMQ 為基礎的 CZMQ[15]之 API 所構成，CZMQ 的製作要點有以下幾項：

- 將 ZeroMQ 的核心部分包裝的更讓人能簡單的了解。

- 隱藏 ZeroMQ API 版本之間的變化。
- 將其中的功能做得更能適合複雜的應用中。
- 使用 High-level Tool 和 API 將 ZeroMQ 中探討的安全考量全部包裝起來。
- 即使其他語言，只要使用 CZMQ 皆能直接作為溝通，更廣泛的跨平台。

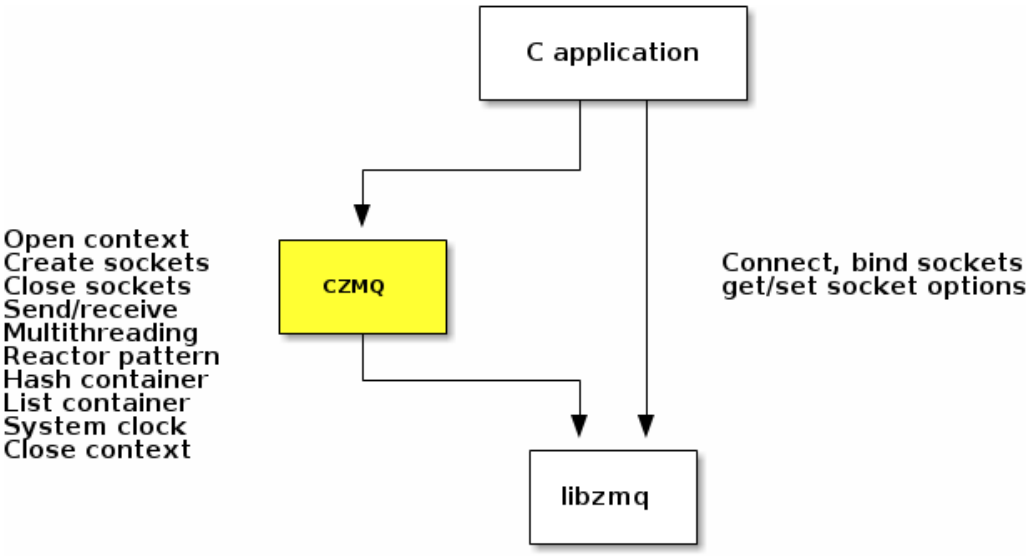


圖2.2　　CZMQ Concept

2.2.3.2 zframe

zframe 為本作品中用來傳輸訊息的最小單位，每一 zframe 都含有一個 zmq_msg_t 的單位，zframe 裡所夾帶的資料主要以 Binary 為主，由 zframe_send ()、zframe_recv () 做收發，可以透過 zframe_more ()來收集項為傳輸完的訊息。

2.2.3.3 zmsg

zmsg 是 ZeroMQ 中所提供用來傳遞訊息的方式之一，組成的最小單位為 zframe，zmsg 以類似 list 的形式，可隨意為欲傳送之訊息增減多個部分，最後再透過 zmq socket 進行傳輸。

2.2.3.4 zhash

由 zmq 所提供可擴充的泛用型 hash table，使用 key 即可拿到對應的資料體，以 1.6 GHz 的電腦下測試，key 長度在 16-char 可有 2.5M/S 的讀取速度。

2.2.3.5 zlist

ZeroMQ 實作的快速泛用型單向 list，可使用之實作多維 list 的應用，甚至和 zhash 一起使用。使用 zlist_first () 取得第一個元素，呼叫 zlist_next () 進行迭代直到最後回傳 NULL 表示結束。

2.2.4 Zthread 線程使用

ZeroMQ 會在背景非同步地處理 I/O 執行緒。可使用 lock-free 資料結構和應用程式執行緒溝通，所以純粹的 ZeroMQ 應用程式不需要 mutex、semaphores 或其他等待狀態就能處理多工環境。

Zthread 提供使用者可如同操作 pthread 般，自由使用來增加需同步處理的項目，並且無須提供 pthread 中較為複雜的 attribute 參數，使用者只須直接使用即可。

2.2.4.1 zthread_fork

可產生一 Attach 的 Thread，會回傳一 Pair Socket 用於和主 Thread 溝通。需給他 zcontex 以處理 Socket 的各種使用變數。

2.2.4.2 zthread_new

用來產生一 Detach 的 Thread，不回傳任何東西模擬了不同 Process 下的運作法，可以產生後就不管其是否結束。Zcontex 和 Attach 的 Thread 不一樣的地方則是有需要再產生就好，並不一定需要。

第三章　系統架構與功能簡介

3.1　系統開發平台

CPU	Intel(R) Pentium(R) Dual CPU T2330 @ 1.60GHz
Memory	2 GB
OS	Ubuntu 13.04
ZeroMQ Version	lbzmq-4.0.3

表3.1　Development Platform

3.2　Watchdog 架構

Watchdog 的主體是由 Frontend 及 Backend 所構成，

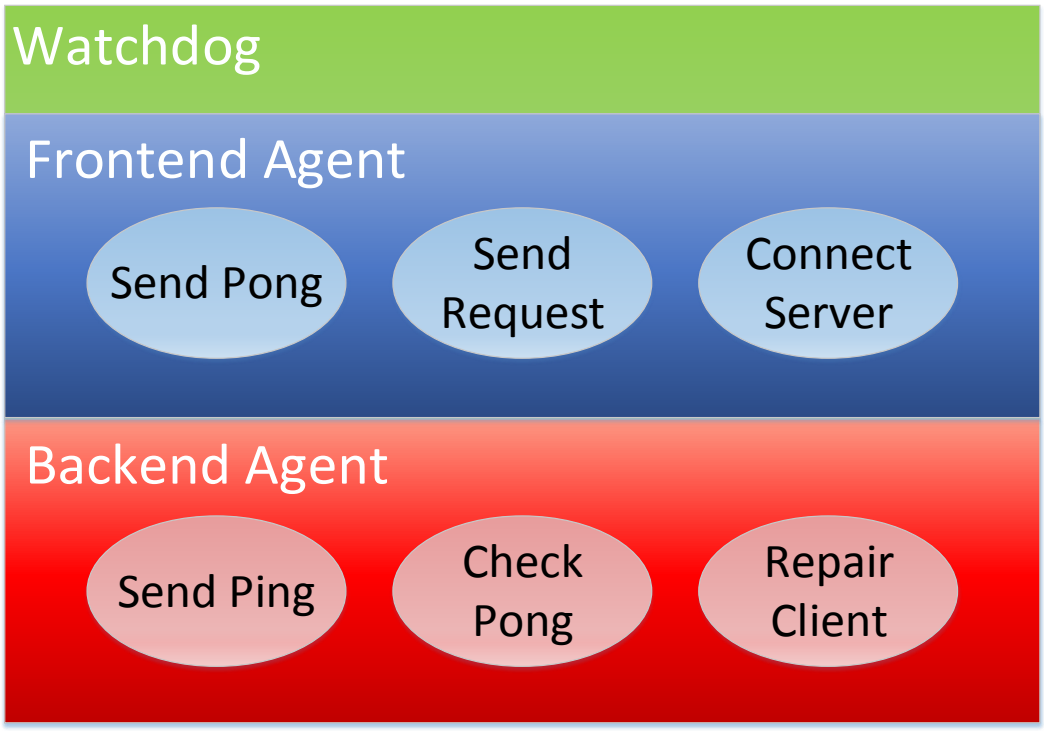


圖3.1　Watchdog main

Frontend：

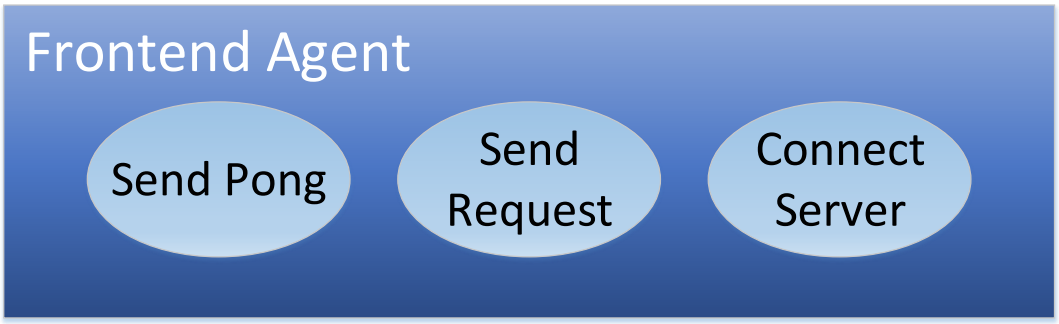


圖3.2　Frontend

Frontend 具有回應 Backend 的能力，透過 zmsg 的包裝在 Ping 送達後可將訊息解開並回應 Pong 回去給監控他的 Backend Server 確保此 Client 依然存在。

並且可透過 Frontend 送出額外的 Request 訊息給在上層的 Backend，其中 Request 訊息主要為 Connect 以及 Virtual Client，

Connect 是 Client 為了連接 Server 必須做的註冊動作；Virtual Connect，並非以 ZeroMQ based 的程式製作，而是負責處理和 Client 有關之其他 Daemon 之回應而成為讓 Backend 誤以為是 Client 的 Virtual Client。

Backend：

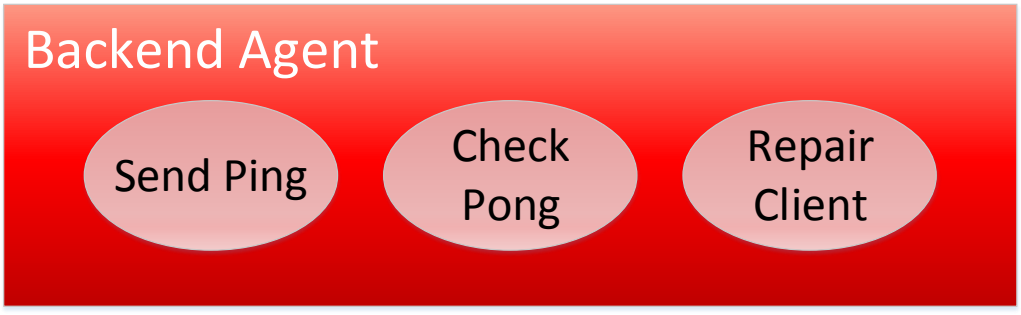


圖3.3　Backend

Backend 需要負責記錄連線過來的任何 Client 或者被註冊過的 Daemon，在正常結束後會留下還在監控的 Client 及 Daemon 名單以便下次啟動時對方不用再次 Connect 就能直接由 Watchdog 叫醒。

目前設定以每秒（Second）作為單位對這些 Client 發送 Ping 或者聯絡 Daemon，等待對方 Frontend 的回覆，或者指定的回覆方式。

最後還需要確定這些目標是否有如期回傳 Pong 或者做回應，如果沒有就需要採取指定的動作，可透過 Configure 設定對一般 Client 採取的動作；由於 Daemon 部分是由使用者在註冊時就決定，所以只需要按照註冊要求執行指令。

依據 Watchdog 中帶有的項目分成三種類型[16][17]，分別是：

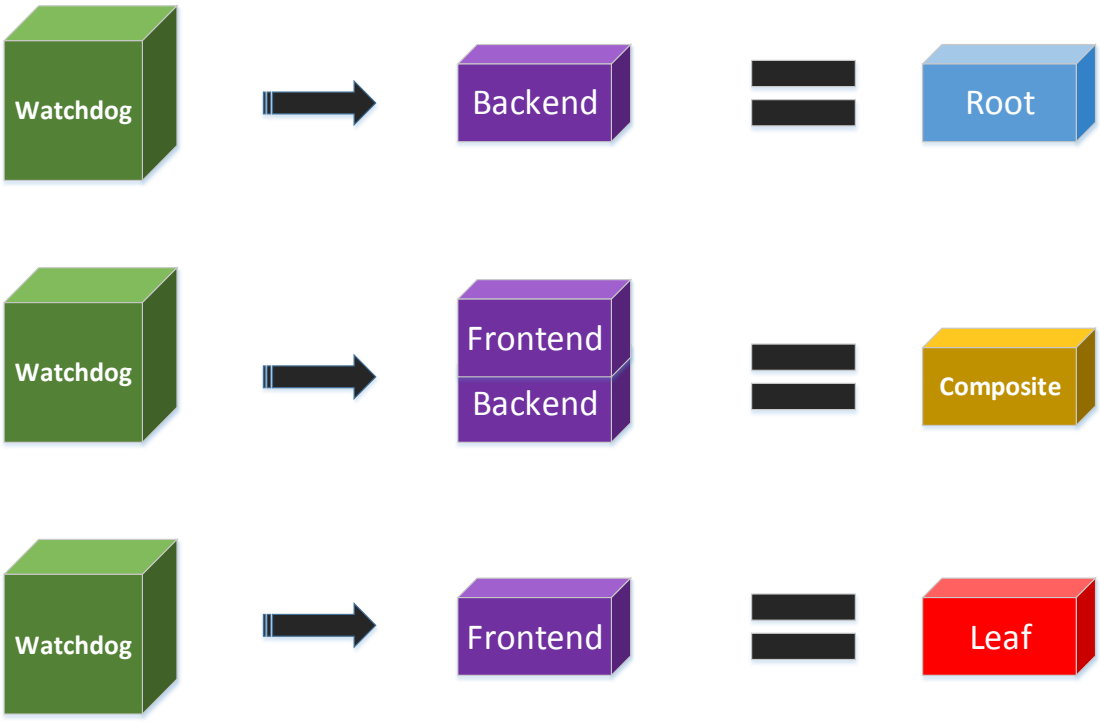


圖3.4　Watchdog structure

將 Watchdog 分類的好處是，如果處於的階級是不需要某一項設計時就能剔除一項負擔，不僅是 CPU 更能省下 Memory 消耗。作為最頂點的已不需要回應 Ping 所以拿掉了 Frontend；作為最末的分支省略 Backend 除了不須處理其他的 Pong 外，還能不打開沒用到的 Port 避免意外發生，ZeroMQ 中只要確定協定、IP、Port 資料，只要用字串就能接通，就安全上的考量能不做就不做。

3.3 Watchdog 監控方式

Watchdog 依種類的分類後的運作情形如下：

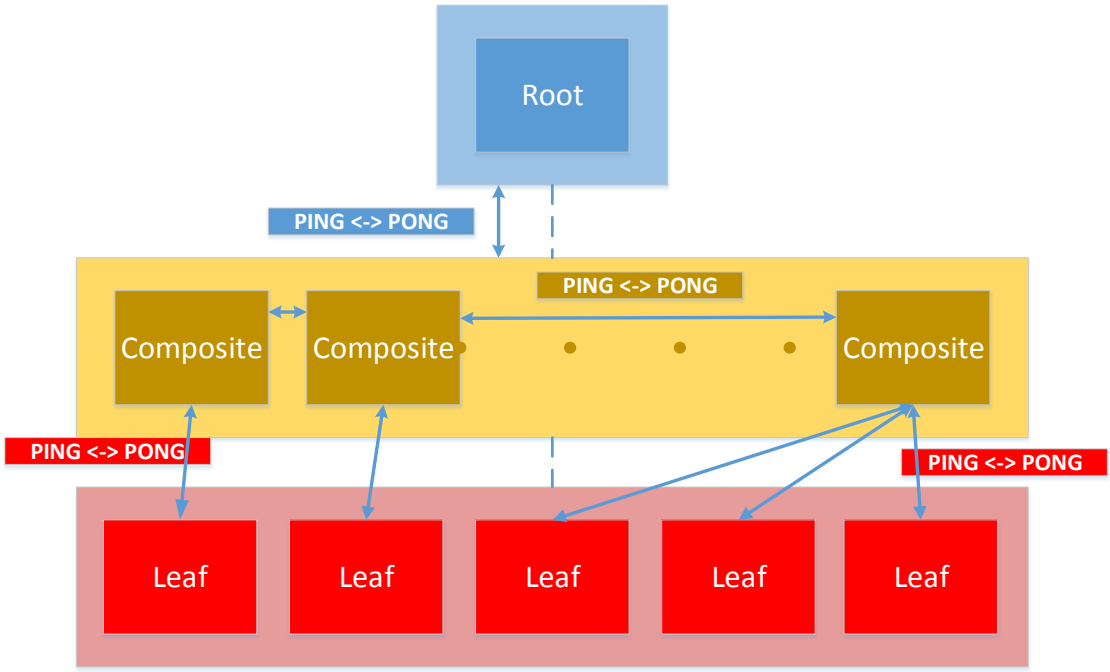


圖3.5　Watchdog life check level

具有唯一的 Root 在他之上不會再有任何 Watchdog，中間可經過數個 Composite 組成甚至可以沒有，最末端點則是由 Leaf 扮演，確定沒有任何需要被監控的 Watchdog。

Root 及 Composite 由於擁有 Backend 底下能再接眾多的 Composite 及 Leaf 而，每一台機器中不一定只能有一個 Watchdog，因此可以依使用需求組成單機或者跨系統的 Watchdog 架構。

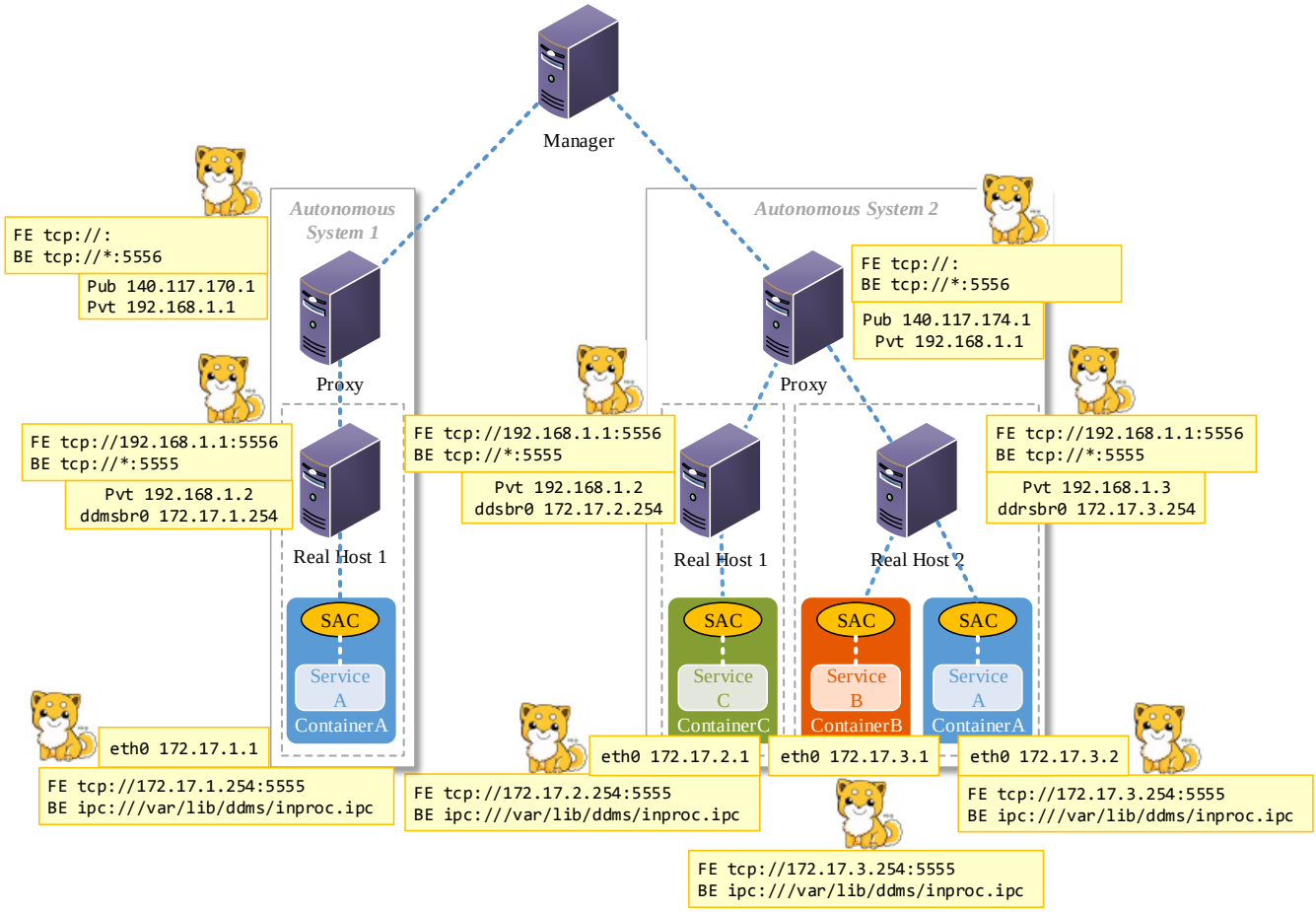


圖3.6　Watchdog in Docker DDMS System

加入了 Watchdog 後，新的架構如圖 3.6，因為 ZeroMQ 的關係，指定 IP 的部分這邊 Backend 寫成 tcp://*:5555 的意義在於，透過 TCP 接收所有來自 Port 5555 的訊息，因此最右端的 Host 可同時監控 Container B 和另一個 Container A；

在上方的 Frontend 則寫為 tcp://192.168.1.1:5556 則是透過 TCP，連接到 192.168.1.1 Port 5556 的位置，所以被他上面的 Proxy 管理著，這樣就符合此架構中的設計模式。

3.3.1 監控說明

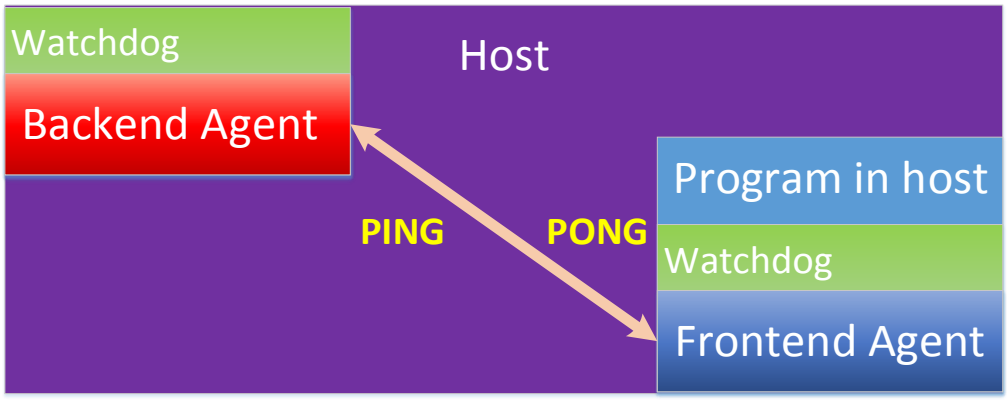


圖3.7　Watchdog ping pong structure

監控中以 PING、PONG 的方式交換訊息確認對方還健在，進行這個動作的固定在兩個不同的 Watchdog 間，並非由同一支 Watchdog 上的 Frontend 及 Backend 互傳訊息，方能建構監視機制。

圖 3.7 中所表示的是在一台 Host 上的 Watchdog（Root）和另一支在 Host 中的某支程式裡的 Watchdog 互通情形。

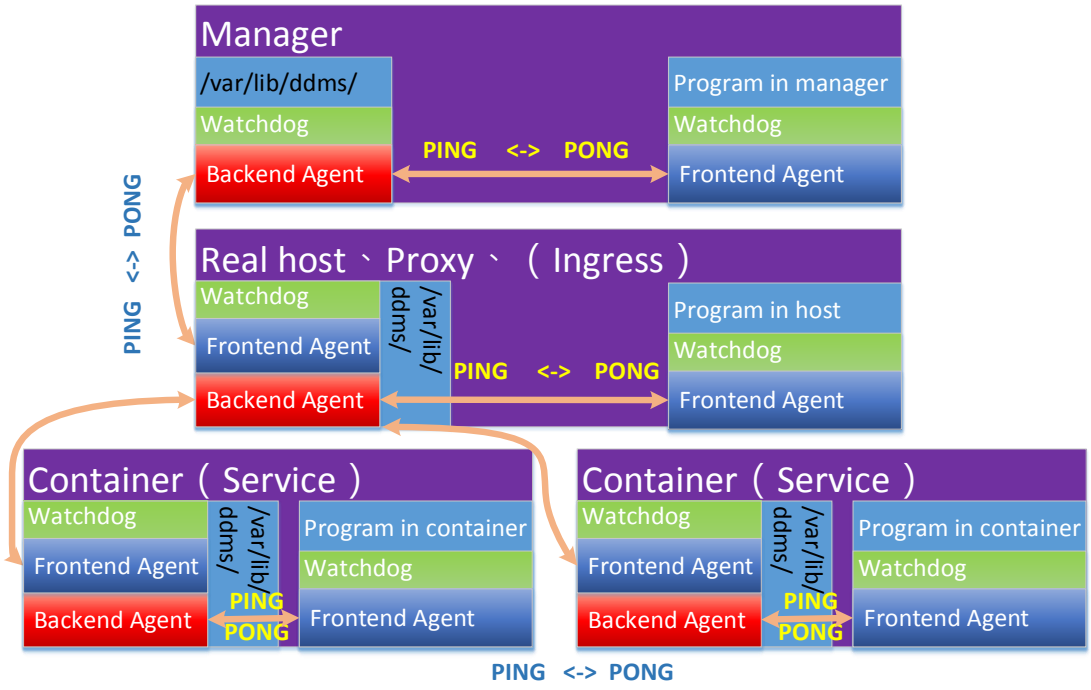


圖3.8　Life check Level in detail

圖 3.8 中則是將 Watchdog 置入由 Docker 所構成的系統中的情形，能更容易地瞭解到將一支 Watchdog 置入/var/lib/ddms 底下，並且讓他和程式中的 Watchdog 溝通可形成單機的監控機制，而底下有其他 Realhost 時則是將 Watchdog 連接的 IP 指定好，就能跨網進行監控。

另一方面在使用者指定的程式中也都直接以 Watchdog（Leaf）通知在自己機器中 Watchdog（Composite）來決定監控對象。

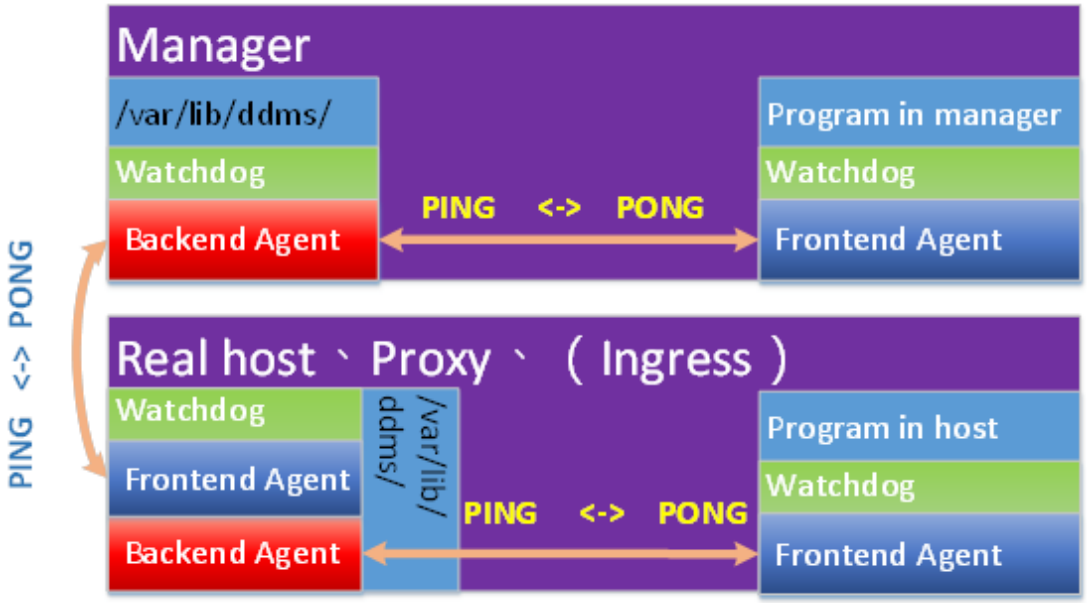


圖3.9　Special watchdog setting

在前面的內容中有說到關於符合條件時，Watchdog 所採取的行動是由 Configure 的內容決定自己該怎麼對除了 Daemon 外的處理方法，但是對於跨機和在自己的程式中雖然都被視為是一般 Client 卻有完全不同的重開方式，這點在現有的架構下只能將 Watchdog 分開使用，一種類型的重開方式需要一種 Watchdog 統一執行。

所以這邊的建議是再使用另一個 Root 或 Composite 的 Watchdog 將這兩種重開法分開。

3.4 ZeroMQ 相關技術

3.4.1 可靠性

常常可以在電腦資訊類科聽到 Reliability 一詞，而這邊會考慮到的要素要如同常聽到一樣：[18]

由於撰寫程式者的功力參差不齊，Code 間可能發生 crash、不正常 exit、freeze、input 突然無法繼續接收資料，程式跑得慢，各種常見的 Coding 問題；而 System Code 依然會因為程式在執行時因為動態環境中的改變，造成 crash、或者因為某些 Client 速度過慢，為了保留訊息最後出現 Memory leak…等情形屢見不鮮。

IPC（Inter Process Communication）中 Message queue 也會 Overflow，當造成溢位時會把多出的訊息過濾掉，所以其他使用者就有可能遺失訊息。

網路出問題的情形已經層出不窮，WiFi 有可能超過範圍收不到訊息，ZeroMQ 對於斷線有做緊急的對應處理，但即使這樣有些訊息還是難免會遺失。

還正在運作重要程式的主機、硬體壞了；因為 Switch 的某些 port 壞掉造成網路出不去、進不來；更嚴重的遇到天災…等各種意想不到的狀況會造成系統不穩定。

要設計出可靠且強健的程式應付上述每一種狀況是不可能達成的，ZeroMQ 為此制定了一些在人為可影響的範圍內還能做的幾個在網路上常見的訊息傳遞方法：

Request-Reply，Client 發現沒辦法拿到回應時會直接放棄並在之後自動重新和 Server 連線，或者找尋其他可用的 Server 做替代方案。而在 Server 端，如果發現 Client 斷線，或者連線不穩定時就會直接判定 Client 已經離開，並將該 Client 清除。

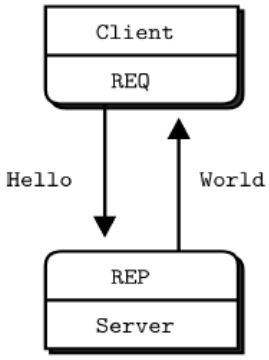


圖3.10　Request-Reply Pattern

Pub-Sub，在此模式中 Server 端並不清楚究竟底下有哪些 Client 已經離開或者死去，Client 也不會將訊息送往 Server，但是當 Client 需要向 Server 訂閱並且要求傳送訂閱之相關訊息時便會透過一 Request-Reply 的 Socket 送出要求 ”重送訂閱之相關內容”的訊息。另外身為訂閱者的 Client 如果發現資紀的速度開始緩慢，也會自動向 Server 發出警告訊息，讓 Server 能更快的採取預防措施。

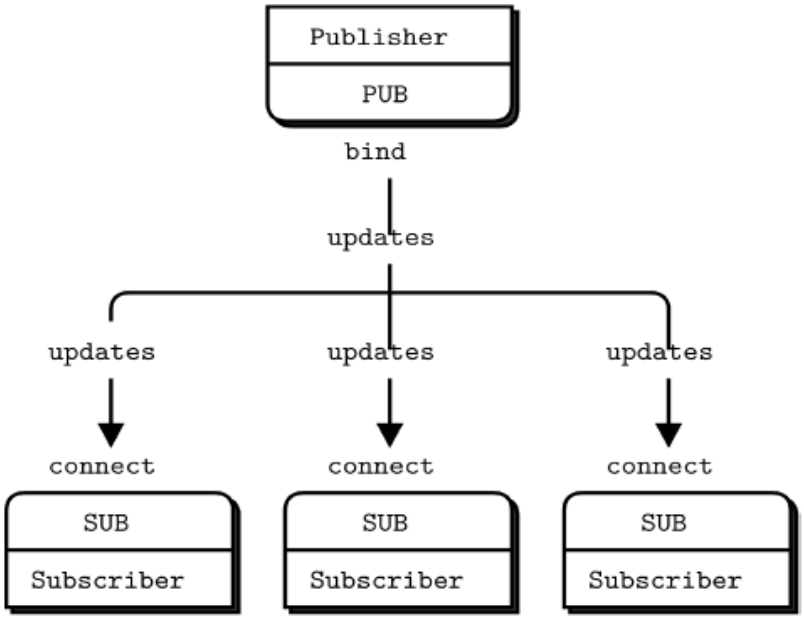


圖3.11　Publish-Subscribe Pattern

Pipeline 中，一個工作的處理方向只有一種路線以防有目的不明確的程序，工作指派者（Ventilator）只會把工作往下交到 Worker 手上，並不曉得底下的工

作是否有順利完成，甚至 Worker 是否還在繼續工作也不清楚。只有在最後面的工作收集者（Collector or Sink）才會知道有哪些工作已完成或未完成，一旦發現沒有完成，才會再往指派者送出要求任務重送的訊息。

如果 Ventilator 或 Sink 的其一毀損，在外面的 Client 無論怎麼丟出工作，也沒辦法得到回應，雖然這部分可能比較不完整，但是確保工作流向的一致性能使網路在處理上的順暢。[5]

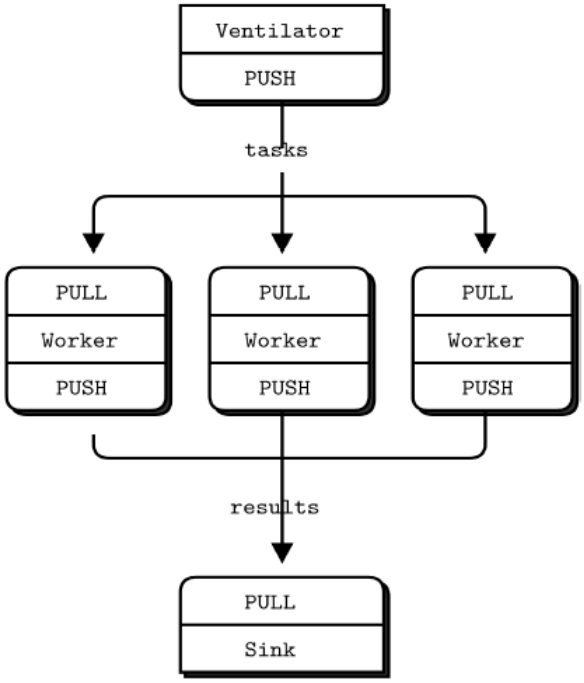


圖3.12　Parallel Pipeline

3.4.2 **Heartbeat**

Hearbeat[19]機制可以掌握點和點之間是否都仍健在，TCP 有個 Timeout 欄位大約可算 30 分鐘，但是這樣一來就沒辦法精確地知道對方是已經死了，或者只是暫時離開那附近。但是 Heartbeat 機制卻也不容易實現，由於單方（或雙方）藉由 Heartbeat 來判定是否斷線，如果因為網路問題而造成 Heartbeat 傳輸失敗，就會產生不應該出的錯，對於 Heartbeat 的使用使用者通常有三種方法：

一、不管他：

通常大部分的狀況下都會這麼作而且只能希望不出錯，ZeroMQ 則不建議這麼作，他會自動將這些連線來的 Peers 隱藏在背後，但這麼作需注意：實作上常使用到 Router Socket 來統一管理這些 Peers，Router Socket 會自動幫忙保留這些 Peers 的資料，但隨著不斷的斷線及連線，終究會產生 Memory leak。

當以 Sub 或 Dealer 為接收端時，無法分辨出沒有資料傳過來到底是因為真的沒有資料或者對方已經不在了，如果他知道對方已經不在，那才能換到另一個 Router 上。

二、Heartbeat 只送一個方向：

雖然可以簡單地認為在 Timeout 之前如果沒收到 Heartbeat 就當作對方已經死了，但是這種概念卻只適用於 Pub-Sub Model，而且也受到諸多限制。有些問題如下：

如果因為傳送大量的資料導致這段時間內都沒送出 Heartbeat，極有可能就因為這樣而 Timeout，不過這一點還可以將任何傳來的資料都視為 Heartbeat 來解決。

Pub-Sub 模式中會自動把要給已經離開的 Subscriber 的訊息丟棄，而 Push 和 Dealer Socket 卻會把這些訊息存起來，如果持續將 Heartbeat 送給一個已離開的使用者，在該使用者回來後將會收到這段時間所有的 Heartbeat。

既然有固定的 Timeout 時間就表示有固定發送 Heartbeat 的頻率，但卻會造成部分使用者如果希望省電而不用這麼常送，或者因為需要很確實的保持 Alive 而縮短這個發送間距。

三、兩邊都送 Heartbeat（Ping-Pong）

只要由一方送出 Ping，對方收到後再回傳 Pong；某些網路上並沒有詳細規定由哪個角色扮演 Client 或 Server，傳送時也不會特定由誰送出 Ping 和 Pong，Timeout 的限制則受到網路拓樸影響，只有 Client 清楚應該是多久，所以通常是由 Client 來送 Ping。

這樣的設計已經適用於 Router Socket 的設計，可參考（二）提到的將任何送過來的資料視為 Pong，沒有資料傳送時再發出 Pong 的方式，會更適合用於其他應用上。

第四章　系統實作

4.1 Watchdog

```
1  pipe = zthread_fork (ctx, watchdog_api_agent, NULL);
```

使用 zthread_fork ()就如同 pthread 可將指定的工作於另一條線程做處理，並且回傳一 ZeroMQ 的 PAIR pipe 回來，使用者就能夠只設計需要的服務，讓 Watchdog 自己處理監控的事物，如果想傳訊息給 Watchdog 就透過回傳的 pipe。

4.2 Frontend

```
1  zmsg_send (送至 Watchdog);
2  zmsg_t *reply = zmsg_recv (self->pipe);
3  if (reply)
4        確認 reply 是正常而非 FAIL。
5  return reply;
```

這部分是保留讓使用者可以發其他的 request 給 Watchdog，如果以後有其他的功能要加，就能直接再往上包裝。

```
1  if (zhash_load (self->configure, "WD_configure") != 0)
```

為了一開始就讓 Watchdog 知道從屬關係，提供 Configure 讓 Watchdog 能按照上面所寫的 IP 連線；上面還包含自己的 ID，Backend 中也需要用到。

```
2  if (streq (command, "CONNECT"))
3        // Do something
4  else if (streq (command, "VIRTUAL-CONNECT"))
5        // Do something
```

```
6   else if (streq (command, "REQUEST"))
7       // Do something
```

Frontend 利用此來分出接下來要送出的是屬於一般的 Client 或是 Daemon 的 Virtual client。

```
1   uint64_t tickless = zclock_time () + 1000 * 3600;
2   if (self->request && tickless > self->expires)
3       tickless = self->expires;
4   int rc = zmq_poll (items, 2, (tickless - zclock_time ()) * ZMQ_POLL_MSEC);
5   if (rc == -1)
6       break;                  //  Context has been shut down
```

ZeroMQ 建議由於這邊可能會有其他的 Request，所以會將 tickless 設定為該 Request 的最後有效時間，所以一直到離最近的 Request 到期時，Frontend 才會起來檢查訊息，以減少 CPU 使用率。

4.3 Backend

```
1   zhash_insert (self->clients, endpoint, client);
2   zhash_freefn (self->clients, endpoint, free_function);
3   zhash_save (self->total_clients, DDMS_HOST_FILENAME);
```

每個連進來的 Client 都會使用 zhash ()紀錄，包含如何將該 Client 在離開時 free 的方法也會先告訴 zhash，然後再將該 zhash 存成檔案以便離開後，還能再找回先前還在監控的 Client。

```
1   if (zhash_load (self->total_clients, DDMS_HOST_FILENAME) == 0)
2           repair_hash (self->total_clients, self->clients, self->virtual_clients);
```

要找回 Client 時使用 zhash_load () 就能將指定的檔案 Load 回 zhash 中，

```
1  sscanf (key, "%[^','], %[^','], %s", expires, ping_command, reboot_command)
```

由於這邊會分為一般的 Client 和 Daemon，需要將格式分出來判斷是監控哪一種的。

```
1  zmq_setsockopt (self->router, ZMQ_IDENTITY, identity, strlen (identity));
```

ZeroMQ 的 Router Socket 需要 ID 所以要先設定，其他人才能透過該 ID 找到對象，如果沒有做這件事，ZeroMQ 會自動給一個 Temp 的 ID 會讓其它要連進來的 Client 無法在一開始就知道 Server 在哪邊。

```
1  zmq_pollitem_t items [] = {
2          { self->pipe    , 0, ZMQ_POLLIN, 0 },
3          { self->router, 0, ZMQ_POLLIN, 0 }
4  };
5  while (!zctx_interrupted)
6  {
7          int rc = zmq_poll (items, 2, 1000 * ZMQ_POLL_MSEC);
8          if (rc == -1)
9                  break;                  //    Context has been shut down
10         if (items [0].revents & ZMQ_POLLIN)
11                 // When message from pipe, it should do…
12         if (items [1].revents & ZMQ_POLLIN)
13                 // When message from router, it should do…
14 }
```

為了能同時處理來自外部及使用者的訊息，Watchdog 內部會使用 zmq_poll () 同時去聽這兩個 ZeroMQ Socket，用法和 poll ()差不多，可以指定時間再檢查，時間還沒到就會先休息把 CPU 釋放出來，就不會造成 while 迴圈一直卡住 CPU 的狀況。

4.4 Watchdog 種類連接實作

```
1  static void manager_agent (void *args, zctx_t *ctx, void *pipe);
2  static void host_agent (void *args, zctx_t *ctx, void *pipe);
3  static void service_agent (void *args, zctx_t *ctx, void *pipe);
```

這邊的三段程式就分別是 Root、Composite、Leaf 要啟動的工作。按種類分開後 Root 只需啟動 Backend，Leaf 則是 Frontend，Composite 需要將兩種都打開以便先後都能連接。

將程式碼分開編譯後就能拿到啟動及監聽項目不同的.o 檔，要編譯出哪種 Watchdog 僅需要編譯時加入不同的部分就能製作完成。

4.5 PING-PONG 機制

Ping & Pong 是很典型在網路上用來確認目標存活的方法，簡單又好用，在 3.4.2 中也有提到以 Ping & Pong 的設計能夠避免在網路即時的環境中，仍須進行解封包以及互相維持 Heartbeat 的工作。

```
1  zhash_foreach (self->clients, client_pong, self->pipe);
2  zhash_foreach (self->clients, client_ping, self->router);
3  zhash_foreach (self->virtual_clients, s_virtual_client_ping, NULL);
4  zhash_foreach (self->virtual_clients, s_virtual_client_pong, NULL);
```

Backend 中會以 foreach ()的方式去跑過登記在 zhash 中所有的 Client，由於 Ping Daemon 的方式不同，foreach ()中的第二個變數可以填入因應不同的 Client 所要採取的不同動作。因此可以對每個 Client 和 Daemon 做 Ping 及收取 Pong 的動作。

```
1  client->ping_at = zclock_time () + PING_INTERVAL;
2  client->expires = zclock_time () + CLIENT_TTL;
```

藉由 zclock_time ()可取得現在的時間，將時間再加上 Ping interval、Expire Time，就能清楚訂出每個 Server 應該回應以及過期的時間；要測定該時間是否已經過了只需再次調用 zclock_time ()，如此就能比較出現在是否已經過了 Time To Live。

```
1  if (streq (command, "PING")) {
2        zmsg_send (回應訊息給從屬的 Watchdog);
3  }
```

當 Frontend 收到從連接的 Watchdog 送來的 PING 訊息後，直接以 PONG 訊息送回給對方，就是最基本的 PING<->PONG 策略。

在 ZeroMQ 的幫助下，已經解決網路系統中因為不穩定而斷線時，會自動重新連線；收到訊息後會先保留在 Queue 中，雖然原本 Unix socket 就已經有了，但是能更有效的利用這些 Queue 內部的資料，達到幾乎不會 Lost 訊息的感覺。

可以相信若偵測到對方沒有連線，就是對方真的已經斷線了，讓我們的設計中能夠撇除網路本身發生的問題，專心看到程式的中心應用部分。

4.6 監控不定性 Daemon 之功能

一般的 Ping-Pong 機制就能處理程式 Fail over 的狀況，但是在現實中卻會遇到程式本身雖然健在也能回應，不過我們需要的 Daemon 或者某個特定功能失效或者沒有在反應時間內回覆，因為本身系統吃重需要處理 Memory…等問題，被我們認定為失效。通常 Watchdog 會使用硬體 Timer 輔助，但是這樣需要

另外再設計硬體對這部分監控，能見到許多單一目的嵌入式系統中用到，既可靠又能減少負擔。

如果這個系統並不是單一目的，可能有復數個部分需要監控，而且可能依據時間有所變化，那這要是直接由硬體寫好未免也太難了，或者增加了成本。

由於 Backend 中使用 ZeroMQ 作為協定，不可能讓每個不一樣的對象都對此協定作特定回應，所以問題是面對這些原本沒有辦法考量的部分該怎麼作才能確定是否需要重開？

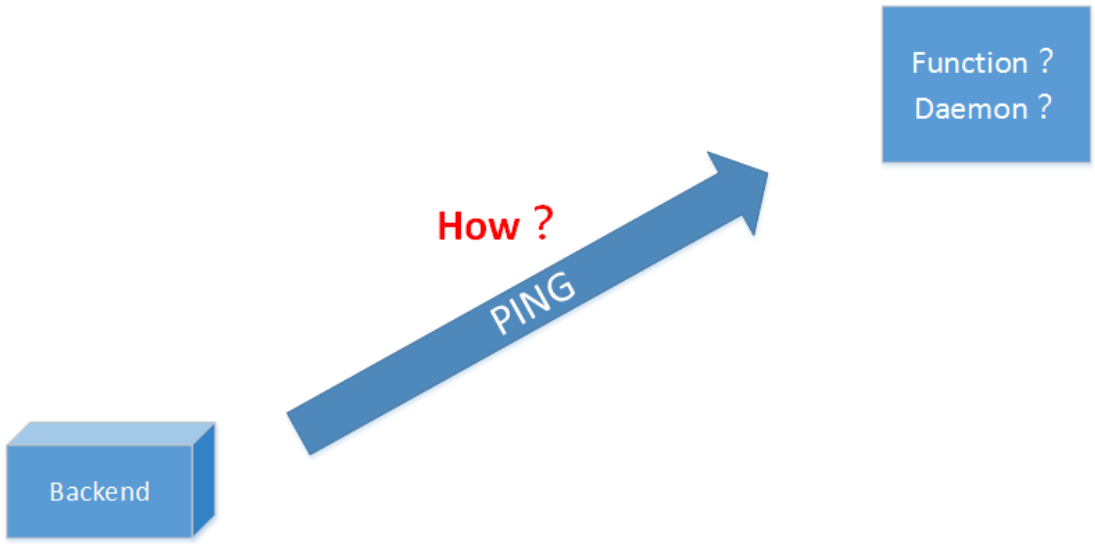


圖4.1　How to ping something not using default way

為此利用了 Backend 中可針對 Client 在固定時間發送 Ping 的特點，由 User 決定要監控的對象、時間作為 Visual Client；關於 Ping 的方式還有當這些已經確定對象不再作用時，如何重開都交由使用者決定。

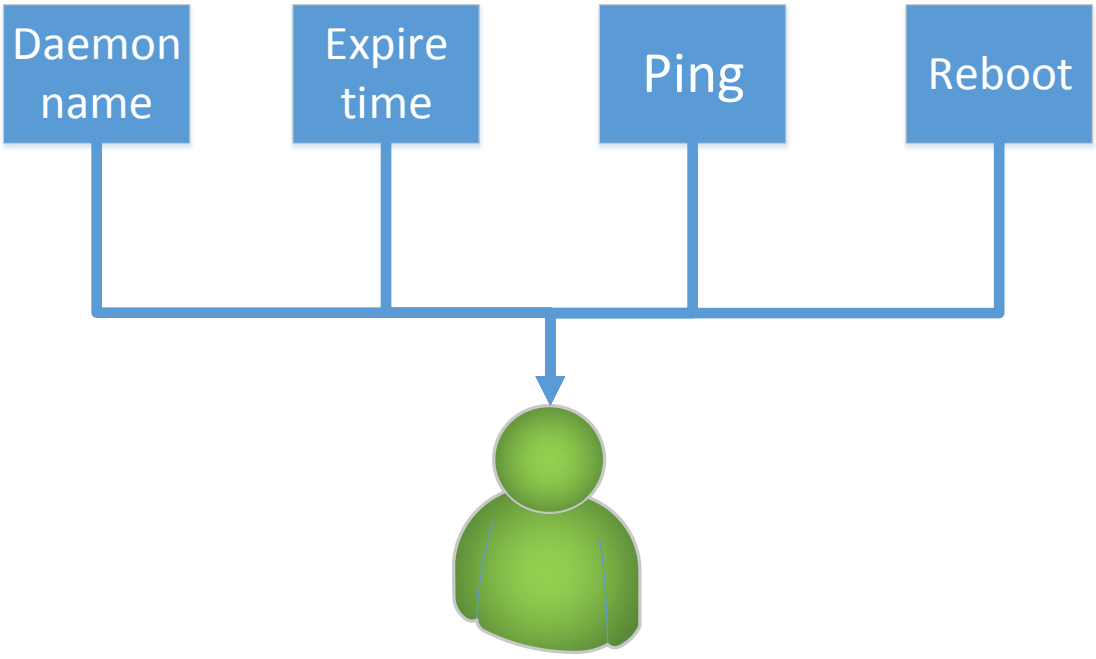


圖4.2　4 variables needed by watchdog watching daemons

4.6.1 函式介紹

使用者可以自行決定是否該使用 Watchdog 的 Daemon 監控，如果希望系統中的 Watchdog 能幫忙，則需呼叫 watchdog_onEye ()函式；若是有其他方法，則不強迫使用，而如果本論文架構的 Watchdog，下方的陣亡必定由上方來通知回復；反之使用者自己寫的復原機制有問題而無法監控目標，那使用者就得自己再想辦法。

watchdog_onEye () 函式需求如下：

```
1  watchdog_onEye (
   daemon name: char*,
   expires in millisecond: unsigned,
   ping source: char* ,
```

```
    reboot source: char*
    )
```

Daemon name：

用來作為 ZeroMQ Router socket 的參考 ID。

Expires：

對 PING 的最晚回應時間，由於各種 Daemon 對要求的回應不一定相同，需要訂出使用者能接受的回應時間，而做 Ping 的動作間隔目前是取 Expire time 的 1/2，也就表示至少在到期前會做出不止一次的 Ping 動作。

ZeroMQ 在設定時會採用 Millisecond (1 sec = 1000 msec) 作為單位，所以輸入時也需要以微秒的單位輸入，而監控的時間也因為這樣可達到秒（Second）以下。

Ping source 和 Reboot source 是填使用者自己寫的 Ping 和 Reboot 程式的 Commands，會把這兩者獨立出來是因為：

對 Ping 而言，要聯絡 Daemon 的方式都大不相同，甚至要考慮需要確認的 Service 所以需要使用者規劃如何 Ping 這個 Daemon，只要成功就讓程式回傳 0，失敗就是-1。

Reboot 通常就是將 Daemon 重開，但是重開的過程中最簡單的除了要重新執行該 Daemon 外，還需要結束原本的 Process。這之間是否需要紀錄相關資訊，或者再開 Daemon 時有些環境參數還需要經過處理後才能使用……等，所有需要的動作都能讓使用者先訂好後再執行。

4.6.2 Ping 的執行流程

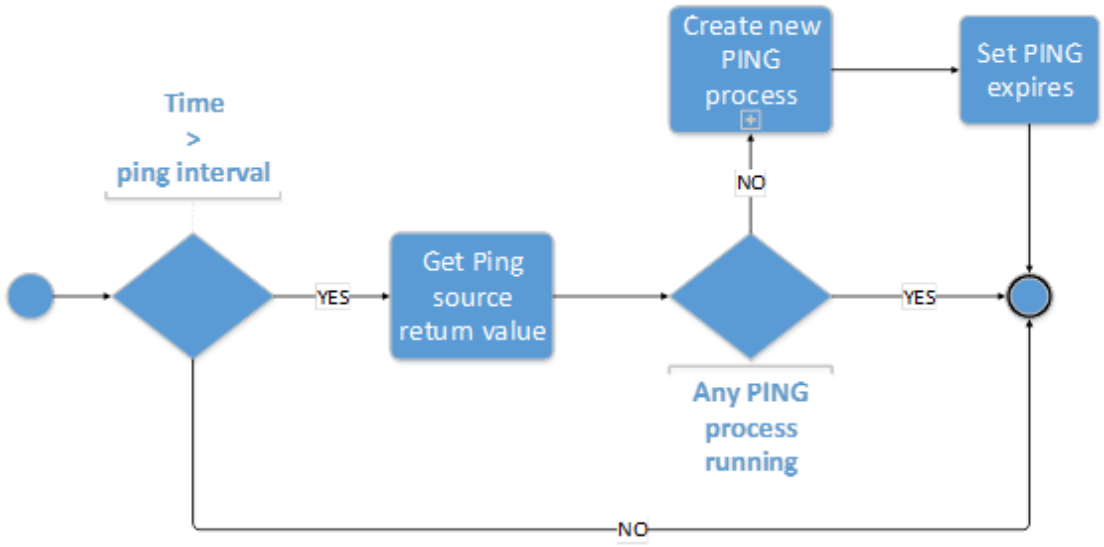


圖4.3　Flow of checking Ping daemon

先檢查是否已經到每次 Ping 的 interval，目前是以使用者填入的 Expire time 的一半作為檢查時間，可以確保在過期前能有一次以上的機會對 Daemon 做 Ping。

一旦確定該檢查時，會把先前 Ping Process 測的結果取出來紀錄目前該 Daemon 的狀態為 Active 或 Inactive，然後確定目前沒有還在運作的 Ping process 後才會再產生新的 Process，一方面避免開啟過多的行程，另一方面也能避開因為多個 Ping 造成連續對結果進行修改。

如果 Process 有延遲，造成原本已經無法運作的 Daemon 狀態又被標成 Active，那就失去 Ping 的意義了。

4.6.3 Reboot 的執行流程

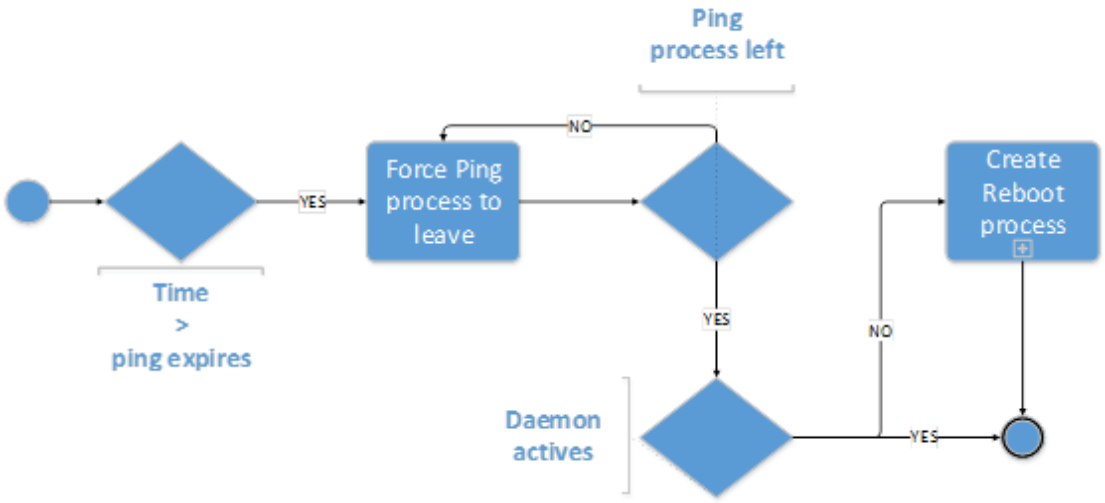


圖4.4　Flow of Reboot daemon

確定已經過了到期時間後，會強制讓目前還在做 Ping 動作的 Process 都先離開，確定不會再對 Daemon 目前的狀態做修改；然後被發現狀態為 Inactive 後，會立即啟動 Reboot 行程進行 Reboot。

4.7　實測結果

```
Dealing Frontend router msg down
Now is 1404873273468
I: Watchdog agent polling time = 3600000
Now is 1404873274469
I: Watchdog agent polling time = 3600000
Dealing Frontend router msg start
Receive message:
-------------------------------------
[003] WTF
[004] PING
Receive message end
Pong message:
-------------------------------------
[003] WTF
[004] PONG
Pong message end
Dealing Frontend router msg down
Now is 1404873275470
I: Watchdog agent polling time = 3600000
```

圖4.5　Watchdog - Frontend Demo

上圖中可以看出 Watchdog 是每 1000 msec 做一次檢查，而且 Frontend 針對 Backend 回應了 PONG 訊息。

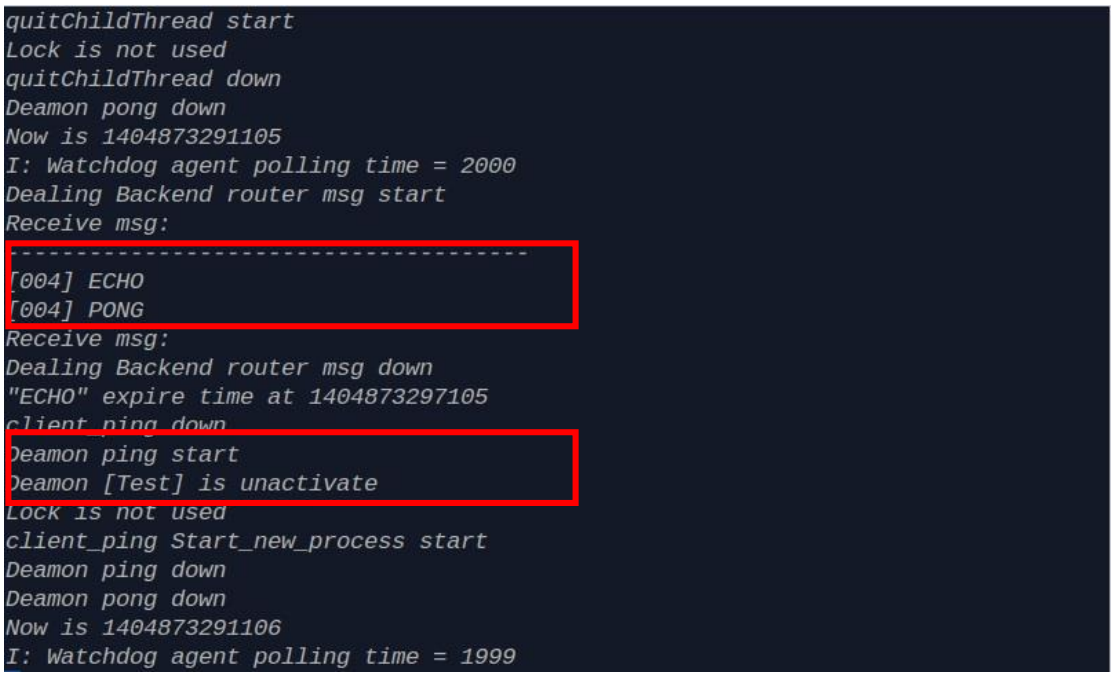


圖4.6　Watchdog -Backend Demo

Backend 除了需要固定 Ping 對下的 Frontend 是否有回傳 PONG 外，還有 Daemon 的狀態需要檢查。

4.8　功能比較

	Linux Watchdog	Linux Supervisor	My Watchdog
錯誤偵測	系統狀態（CPU、Memory、Temperatures…）、Program 持續寫入	程式狀態（RUNNING、STOPPED…）	PING-PONG 訊息、自定義 Ping 方式
錯誤處置	Reboot System	Restart desire program	Reboot System & Restart desire program

監控層級	Kernel-Space &User-Space	User-Space	User-Space
監控範圍	單機	單機	單機 & 跨網
使用方式	Configure & 檔案 I/O	Script	API
使用難度	難	易	易
使用限制	硬體 & Kernel 支援	程式需跑在前景	能裝 ZeroMQ library & CZMQ Library
使用平台	Linux	Linux	跨平台（以 ZeroMQ 支援為主）

表4.1 Comparsion between three kind of watchdogs

我們將 Watchdog 難以使用的部分簡化，並且提供可跨網的特性，又因為虛擬機上的系統有可能不同，在實作時選擇 ZeroMQ 同時為了網路及可跨平台的特性。

雙向的 Ping-Pong 機制已經足以發揮監控對方的能力；Reboot 的部分則是通過 configure 設定 Reboot 方式，因此能做到重開系統或者重開程式的分別。

第五章　結論

網路本身就是不穩定的通道，但在近期的網路品質不斷地提高下也讓原先電腦一定要有 OS 的概念漸漸改變，各大廠所提供的虛擬系統平台租用服務成了現在的局勢之一，隨著劇增這些系統的使用率就考驗著提供者伺服器的能力，消費者希望盡可能少租用服務換取最大利用率，而系統的可靠程度也會成為他們評斷的標準，不僅是程式的設計功力，就連即使發生錯誤後是否能在最快的速度下回復服務也會對雙方造成很大的影響。

ZeroMQ 針對網路程式中常遇到的問題即使不完美，卻也幫了程式撰寫者不少的忙，不需要考慮同步、連線過程……等問題。在非常忙碌的狀況下也能保持低 CPU 的使用率，程式的重點漂亮的集中於需要做的事而非在網路上的其他問題。

5.1　目標 & 問題

無法使用 Linux Watchdog & Linux Supervisor 來達到我們本篇的目的是在於即使能將這兩者應用於跨網的設計中，也無法達到跨平台的需求，這會將虛擬系統受限在同樣的作業系統中，造成很不方便的使用方式。

在本篇中運用 ZeroMQ 強大的網路程式處理能力實作了能夠跨網的 Watchdog 系統，由於實機可能存在於各地的虛擬系統中，有時候各台機器所使用的作業系統也可能不同，Watchdog 在這種情況下非常需要能夠克服網路障礙，並且支援多種平台，給予管理者適當的回報。

在實作過程中最大的問題在於協定的制訂上，雖然 ZeroMQ 完整支援了常見的協定，雙方再互送資料時有可能因為協定上寫錯了格式就發生 zmsg 寄了卻完全沒反應的情形；

其次是連線的同步，ZeroMQ 確實能做到相當快的連線建立，但是卻因為他太快了，程式中在建立連線後不能法上進行傳輸，預設提供 zclock_sleep () 能確定讓 ZeroMQ 建立連線並且不佔用 CPU 的使用率，和系統函式的 sleep () 會暫時讓 process 停下來的意義有些不同；

經過測試發現，即使以上都做到後依然有可能會發生沒有連線到的狀況，只有這一點無法確定究竟是哪邊出了問題，因為通常在連續重開約 1 ～ 2 次後，就能正常連線，有可能是 Bind 方沒有準備好，這點可能還有待釐清。

5.2 未來展望

在未來的改進中，本篇於單一 Process 使用了許多結構，在不斷的測試後發現 Memory 很有可能因此不夠，為了能夠讓使用者能更將焦點停留在自己的程式而不是減少不必要的 Memory 消耗上，考慮應該將可能造成耗損的監視 Daemon 部分分別到其他 Process 執行；另一點是需要將此 Watchdog 編成函式庫的形式，讓使用者能夠以 Device 的方式呼叫將會更有利於使用。

相信能跨網的 Wathcdog 在這種雲類型的系統中一定能發揮他應有的功能，他雖然只是系統的一部份，卻能提供系統更強的可靠度，在任何應用上都會成為很大的助力。

參考文獻

[1] dotCloud, “Docker - Build, Ship and Run Any App, Anywhere”, https://www.docker.com/

[2] Robertson, Alan. "The evolution of the Linux-HA project." UKUUG LISA/Winter Conference High-Availability and Reliability. 2004.

[3] Pieter Hintjens, “ZeroMQ Messaging for Many Applications”, 2013\03

[4] Addison-Wesley, “Advanced Programming in the UNIX Environment”, 1992.

[5] Soltesz, Stephen, et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.

[6] Abaffy, J., & Krajcovic, T. (2010, September). Software support for multiple hardware watchdog timers in the Linux OS. In Applied Electronics (AE), 2010 International Conference on applied electronics, pp. 1-3

[7] Supervisor: A Process Control System, “Supervisor Documentation”, http://supervisord.org/index.html

[8] Christian's Blog, “Using the Watchdog Timer in Linux”, http://www.jann.cc/2013/02/02/linux_watchdog.html, February 02, 2013

[9] kunilkuda, “How to Use Linux Watchdog”, http://embeddedfreak.wordpress.com/2010/08/23/howto-use-linux-watchdog/, August 23, 2010

[10] “How To Install and Manage Supervisor on Ubuntu and Debian VPS”, https://www.digitalocean.com/community/tutorials/how-to-install-and-manage-supervisor-on-ubuntu-and-debian-vps

[11] RADULOVIC, Alex. Private IP communication network architecture. U.S. Patent No 7,215,663, 2007.

[12] NORDMAN, Mikael. Secure access method, and associated apparatus, for accessing a private IP network. U.S. Patent No 6,061,346, 2000.

[13] Addison-Wesley, "Advanced Programming in the UNIX Environment Second edition", pp. 545-584, 1992

[14] Free Software Foundation, "The GNU Lesser General Public License, version 3.0", http://opensource.org/licenses/LGPL-3.0

[15] iMatix Corporation, "CZMQ - High-level C Binding for ZeroMQ", http://czmq.zeromq.org/

[16] RIEHLE, Dirk. Composite design patterns. In: ACM SIGPLAN Notices. ACM, 1997. pp. 218-228.

[17] GAMMA, Erich, et al. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[18] Pieter Hintjens, published by iMatix, "0MQ Code Connected Volume 1" http://hintjens.wdfiles.com/local--files/main:files/cc1pe.pdf, pp. 135-137

[19] Pieter Hintjens, published by iMatix, "0MQ Code Connected Volume 1" http://hintjens.wdfiles.com/local--files/main:files/cc1pe.pdf, pp. 152-153

[20] LÓ PEZ-PÉREZ, David; GUVENC, Ismail; CHU, Xiaoli. Theoretical analysis of handover failure and ping-pong rates for heterogeneous networks. (ICC), 2012 IEEE International Conference on Communications. IEEE, 2012. pp. 6774-6779.

[21] DOLEV, Danny; EVEN, Shimon; KARP, Richard M., "On the security of ping-pong protocols." Information and Control, 1982, 55.1: pp. 57-68.

[22] SUN, Mingqiu; TONN, Jeffrey. Network health monitoring through real-time analysis of heartbeat patterns from distributed agents. 2001. Pub. No.: US 2003/0061340 A1

[23] MCINTYRE, Michael S., et al. “Network controller system that uses multicast heartbeat packets.” U.S. Patent No 6,272,113, 2001.

[24] DARWIN, Sam; FALCO, Vincent; NICPONSKI, Dave. Peer-to-peer network heartbeat server and associated methods. U.S. Patent Application 10/881,570, 2004.

[25] LE, Hung; TENE, Gil. Server fail-over system. U.S. Patent No 6,145,089, 2000.

[26] GADIR, Omar MA, et al. High-availability cluster virtual server system. U.S. Patent No 6,944,785, 2005.

[27] GRAY, Jim; SIEWIOREK, Daniel P. High-availability computer systems. Computer, 1991, 24.9: pp. 39-48.

[28] Hoi Chan and Trieu Chieu. 2012. An approach to high availability for cloud servers with snapshot mechanism. In Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE '12). ACM, New York, NY, USA, Article 6, 6 pages.

[29] Yihan Wu and Gang Huang. 2013. Model-based high availability configuration framework for cloud. In Proceedings of the 2013 Middleware Doctoral Symposium (MDS '13). ACM, New York, NY, USA, Article 6, 6 pages.

[30] Kanwardeep Singh Ahluwalia and Atul Jain. 2006. High availability design patterns. In Proceedings of the 2006 conference on Pattern languages of programs (PLoP '06). ACM, New York, NY, USA, Article 19, 9 pages.

[31] Balazs Gerofi and Yutaka Ishikawa. 2012. Enhancing TCP throughput of highly available virtual machines via speculative communication. In Proceedings of the

8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12). ACM, New York, NY, USA, pp. 87-96.

[32] Michael Pearce, Sherali Zeadally, and Ray Hunt. 2013. Virtualization: Issues, security threats, and solutions. ACM Comput. Surv. 45, 2, Article 17 (March 2013), 39 pages.

[33] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. 2010. The design of a practical system for fault-tolerant virtual machines. SIGOPS Oper. Syst. Rev. 44, 4 (December 2010), pp. 30-39.

[34] Oreste Villa, Sriram Krishnamoorthy, Jarek Nieplocha, and David M. Brown, Jr.. 2009. Scalable transparent checkpoint-restart of global address space applications on virtual machines over infiniband. In Proceedings of the 6th ACM conference on Computing frontiers (CF '09). ACM, New York, NY, USA, pp. 197-206.

[35] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting failures in distributed systems with the Falcon spy network. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY, USA, pp. 279-294

[36] Mark L. McKelvin, Jr., Gabriel Eirea, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. 2005. A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In Proceedings of the 5th ACM international conference on Embedded software (EMSOFT '05). ACM, New York, NY, USA, pp. 237-246.

[37] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. 2010. On the impact of fault tolerance tactics on architecture patterns. In Proceedings of the 2nd International

Workshop on Software Engineering for Resilient Systems (SERENE '10). ACM, New York, NY, USA, pp. 12-21.

[38] Yuriy Brun and Nenad Medvidovic. 2007. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In Proceedings of the 2007 workshop on Engineering fault tolerant systems (EFTS '07). ACM, New York, NY, USA, , Article 7 .

[39] Alexander B. Romanovsky. 1992. Synchronization as a framework for distributed system fault-tolerance design. In Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring (EW 5). ACM, New York, NY, USA, 1-5.

[40] Ozgur Koray Sahingoz and A. Coskun Sonmez. 2006. Fault tolerance mechanism of agent-based distributed event system. In Proceedings of the 6th international conference on Computational Science - Volume Part III (ICCS'06), Vassil N. Alexandrov, Geert Dick Albada, Peter A. Sloot, and Jack Dongarra (Eds.), Vol. Part III. Springer-Verlag, Berlin, Heidelberg,

附錄

在虛擬系統上之 Watchdog 設計與實作

[1]林俊宏　[2]鄭又新

國立中山大學資訊工程學系

[1] lin@cse.nsysu.edu.tw, [2]david2274627@gmail.com

摘要

Docker[1]是個強大的虛擬機器配置系統，使用者可以依照使用需求，透過簡單的指令去配置虛擬機器－Container；讓客戶想要執行的程式放在底下作為一個 Turnkey，然後再讓我們能以 Turnkey 為單位統一管理這些虛擬裝置。

如果這些運行的系統發生故障，想必會造成很大的問題，因此也需要能夠監視這套 Docker 系統（包含 Manager、Ingress、Proxy、Real host、Containers）的 Watchdog，以便於在發生狀況時能及時搶救，免於更大的損失。

1. 前言

在這套 Watchdog 中，我們以 ZeroMQ[2]作為連線的工具，和一般 UNIX Domain Socket[3]相比 ZeroMQ 已經模擬了現實生活中許多的連線狀況，並且能處理大部分的問題。

為了方便使用者能在設計服務時使用，本論文會將此 Watchdog 做成 API 的形式來呈現。

2. 相關研究

2.1 Linux Watchdog

Linux 系統上具備了的 Hardware Watchdog 和 Software Watchdog[5]，部分硬體設計時就已經支援 Watchdog 的機制，另一部分透過 Kernel 軟體支援成為 Watchdog module 使用者可在安裝 Watchdog Daemon 後使用，無論是哪一種用法上都是先開啟 /dev/watchdog 後持續對此裝置寫入，將 watchdog timer 重置，如果未在期限內繼續寫入，或者發生了不符合 configure 中的條件，Watchdog 即會立即將系統重開；最後使用者可以規定的形式關閉 Watchdog，但是如果編譯 Kernel 時將 nowayout 參數開啟（CONFIG_WATCHDOG_NOWAYOUT），Watchdog 就無法被其他方式關閉。

2.2 Linux Supervisor

Linux Supervisor[4]只有提供 Unix based System 使用，主要用來做 Processes 間的管理要務。為了操作 Supervisor 可以進入 supervisorctl 中查看目前監控下的 Process 狀態，這邊主要分 RUNNING、STOPPED 兩種，注意：若需要 Supervisor 幫忙觀察則 Program 必須是執行於前景，否則無法正確地

知道 program 的當前狀態，更可透過 start、restart、stop 這些指令改變目前監控的項目應該表示的狀態。

Supervisor 使用簡單並且能掌握 Process 間的狀況，對於不是需要相當重要的系統性的 Reboot 而只是 User 自行定義的一些程式而言，可說是既方便又實在。

3. 系統架構與功能簡介

3.1 Watchdog 架構

Watchdog 的主體是由 Frontend 及 Backend 所構成，

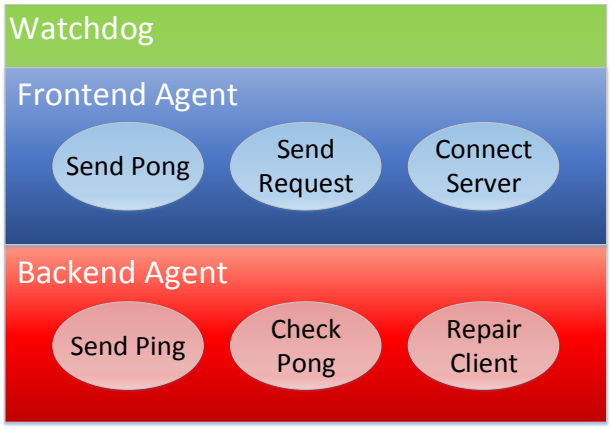


圖3.1 Watchdog main

Frontend 具有回應 Backend 的能力，透過 zmsg 的包裝在 Ping 送達後可將訊息解開並回應 Pong 回去給監控他的 Backend Server 確保此 Client 依然存在。

並且可透過 Frontend 送出額外的 Request 訊息給在上層的 Backend，其中 Request 訊息主要為 Connect 以及 Virtual Client，

Connect 是 Client 為了連接 Server 必須做的註冊動作；Virtual Connect，並非以 ZeroMQ based 的程式製作，而是負責處理和 Client 有關之其他 Daemon 之回應而成為讓 Backend 誤以為是 Client 的 Virtual Client。

Backend 需要負責記錄連線過來的任何 Client 或者被註冊過的 Daemon，在正常結束後會留下還在監控的 Client 及 Daemon 名單以便下次啟動時對方不用再次 Connect 就能直接由 Watchdog 叫醒。

目前設定以每秒（Second）作為單位對這些 Client 發送 Ping 或者聯絡 Daemon，等待對方 Frontend 的回覆，或者指定的回覆方式。

最後還需要確定這些目標是否有如期回傳 Pong 或者做回應，如果沒有就需要採取指定的動作，可透過 Configure 設定對一般 Client 採取的動作；由於 Daemon 部分是由使用者在註冊時就決定，所以只需要按照註冊要求執行指令。

依據 Watchdog 中帶有的項目分成三種類型[17][18]，分別是：

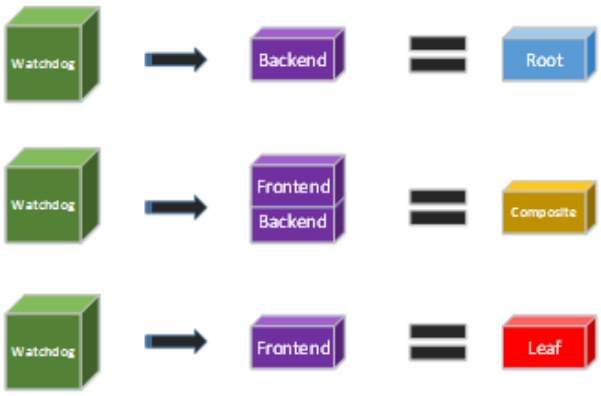


圖3.2 Watchdog structure

將 Watchdog 分類的好處是，如果處於的階級是不需要某一項設計時就能剔除一項負擔，不僅是 CPU 更能省下 Memory 消耗。作為最頂點的已不需要回應 Ping 所以拿掉了 Frontend；作為最末的分支省略 Backend 除了不須處理其他的 Pong 外，還能不打開沒用到的 Port 避免意外發生，ZeroMQ 中只要確定協定、IP、Port 資料，只要用字串就能接通，就安全上的考量能不做就不做。

3.2 Watchdog 監控方式

Watchdog 依種類的分類後的運作情形如下：

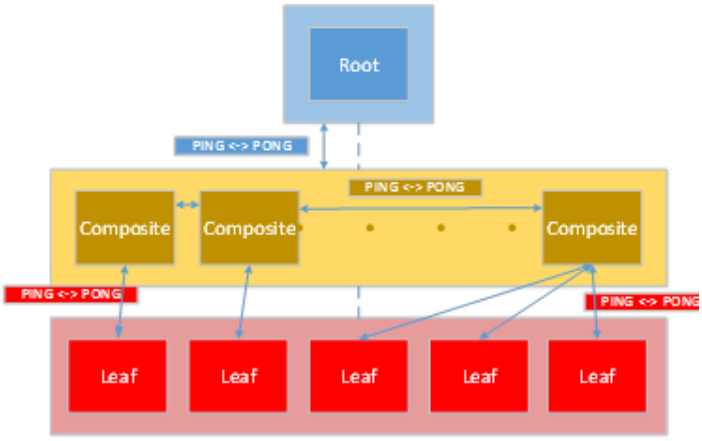


圖3.3 Watchdog life check level

具有唯一的 Root 在他之上不會再有任何 Watchdog，中間可經過數個 Composite 組成甚至可以沒有，最末端點則是由 Leaf 扮演，確定沒有任何需要被監控的 Watchdog。

Root 及 Composite 由於擁有 Backend 底下能再接眾多的 Composite 及 Leaf 而，每一台機器中不一定只能有一個 Watchdog，因此可以依使用需求組成單機或者跨系統的 Watchdog 架構。

```
FE tcp://192.168.1.1:5556
BE tcp://*:5555
```

圖3.4 Watchdog 示意圖

因為 ZeroMQ 的關係，指定 IP 的部分這邊 Backend 寫成 tcp://*:5555 的意義在於，透過 TCP 接收所有來自 Port 5555 的訊息，在上方的 Frontend 則寫為 tcp://192.168.1.1:5556 則是透過 TCP，連接到 192.168.1.1 Port 5556 的位置。

3.3 監控說明

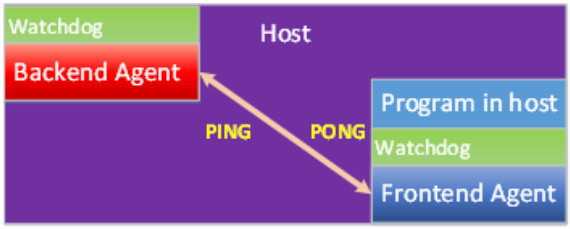


圖3.5 Watchdog ping pong structure

這是在一台 Host 上的 Watchdog（Root）和另一支在 Host 中的某支程式裡的 Watchdog 互通情形。

監控中以 PING、PONG 的方式交換訊息確認對方還健在，進行這個動作的固定在兩個不同的 Watchdog 間，並非由同一支 Watchdog 上的 Frontend 及 Backend 互傳訊息，方能建構監視機制。

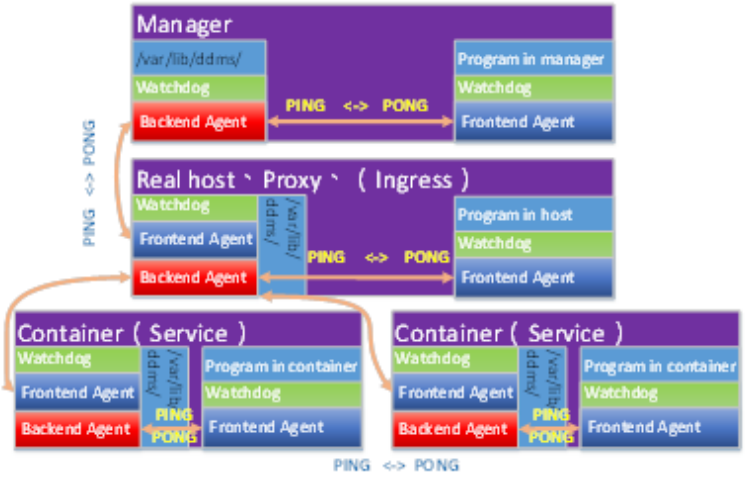


圖3.6 Life check Level in detail

圖 3.6 中則是將 Watchdog 置入一由 Docker 所構成的系統中的情形，能更容易地瞭解到將一支 Watchdog 置入/var/lib/ddms 底下，並且讓他和程式中的 Watchdog 溝通可形成單機的監控機制，而底下有其他 Realhost 時則是將 Watchdog 連接的 IP 指定好，就能跨網進行監控。

另一方面在使用者指定的程式中也都直接以 Watchdog（Leaf）通知在自己機器中 Watchdog（Composite）來決定監控對象。

3.4 監控不定性 Daemon 之功能

一般的 Ping-Pong 機制就能處理程式 Fail over 的狀況，但是在現實中卻會遇到程式本身雖然健在也能回應，不過我們需要的 Daemon 或者某個特定功能失效或者沒有在反應時間內回覆，因為本身系統吃重需要處理 Memory…等問題，被我們認定為失效。通常 Watchdog 會使用硬體 Timer 輔助，但是這樣需要另外再設計硬體對這部分監控，能見到許多單一目的嵌入式系統中用到，既可靠又能減少負擔。

使用者可以自行決定是否該使用 Watchdog 的 Daemon 監控，如果希望系統中的 Watchdog 能幫忙，則需呼叫 watchdog_onEye ()函式；若是有其他方法，則不強迫使用，而如果本論文架構的 Watchdog，下方的陣亡必定由上方來通知回復；反之使用者自己寫的復原機制有問題而無法監控目標，那使用者就得自己再想辦法。

watchdog_onEye () 函式需求如下：

watchdog_onEye (

daemon name: char*,

expires in millisecond: unsigned,

ping source: char* ,

reboot source: char*

)

Daemon name：

用來作為 ZeroMQ Router socket 的參考 ID。

Expires：

對 PING 的最晚回應時間，由於各種 Daemon 對要求的回應不一定相同，需要訂出使用者能接受的回應時間，而做 Ping 的動作間隔目前是取 Expire time 的 1/2，也就表示至少在到期前會做出不止一次的 Ping 動作。

ZeroMQ 在設定時會採用 Millisecond (1 sec = 1000 msec) 作為單位，所以輸入時也需要以微秒的單位輸入，而監控的時間也因為這樣可達到秒（Second）以下。

Ping source 和 Reboot source 是填使用者自己寫的 Ping 和 Reboot 程式的 Commands，會把這兩者獨立出來是因為：

對 Ping 而言，要聯絡 Daemon 的方式都大不相同，甚至要考慮需要確認的 Service 所以需要使用者規劃如何 Ping 這個 Daemon，只要成功就讓程式回傳 0，失敗就是-1。

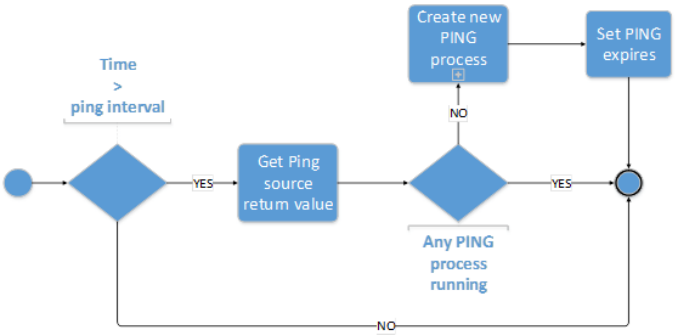


圖3.7 Flow of checking Ping daemon

先檢查是否已經到每次 Ping 的 interval，目前是以使用者填入的 Expire time 的一半作為檢查時間，可以確保在過期前能有一次以上的機會對 Daemon 做 Ping。

一旦確定該檢查時，會把先前 Ping Process 測的結果取出來紀錄目前該 Daemon 的狀態為 Active 或 Inactive，然後確定目前沒有還在運作的 Ping process 後才會再產生新的 Process，一方面避免開啟過多的行程，另一方面也能避開因為多個 Ping 造成連續對結果進行修改。

如果 Process 有延遲，造成原本已經無法運作的 Daemon 狀態又被標成 Active，那就失去 Ping 的意義了。

Reboot 通常就是將 Daemon 重開，但是重開的過程中最簡單的除了要重新執行該 Daemon 外，還需要結束原本的 Process。這之間是否需要紀錄相關資訊，或者再開 Daemon 時有些環境參數還需要經過處理後才

能使用……等，所有需要的動作都能讓使用者先訂好後再執行。

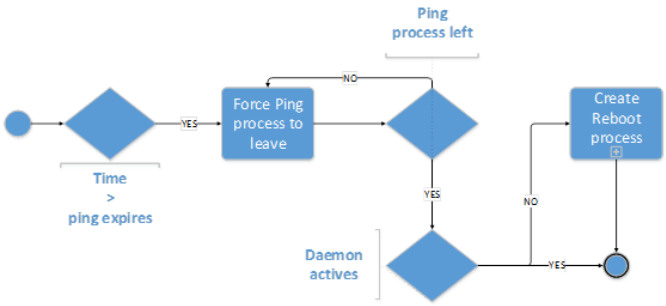


圖3.8 Flow of Reboot daemon

確定已經過了到期時間後，會強制讓目前還在做 Ping 動作的 Process 都先離開，確定不會再對 Daemon 目前的狀態做修改；然後被發現狀態為 Inactive 後，會立即啟動 Reboot 行程進行 Reboot。

3.5 實測結果

```
Dealing Frontend router msg down
Now is 1404873273468
I: Watchdog agent polling time = 3600000
Now is 1404873274469
I: Watchdog agent polling time = 3600000
Dealing Frontend router msg start
Receive message:
-----------------------------------
[003] WTF
[004] PING
Receive message end
Pong message:
-----------------------------------
[003] WTF
[004] PONG
Pong message end
Dealing Frontend router msg down
Now is 1404873275470
I: Watchdog agent polling time = 3600000
```

圖3.9 Watchdog - Frontend Demo

上圖中可以看出 Watchdog 是每 1000 msec 做一次檢查，而且 Frontend 針對 Backend 回應了 PONG 訊息。

```
quitChildThread start
Lock is not used
quitChildThread down
Deamon pong down
Now is 1404873291105
I: Watchdog agent polling time = 2000
Dealing Backend router msg start
Receive msg:
-----------------------------------
[004] ECHO
[004] PONG
Receive msg:
Dealing Backend router msg down
"ECHO" expire time at 1404873297105
client_ping down
Deamon ping start
Deamon [Test] is unactivate
Lock is not used
client_ping Start_new_process start
Deamon ping down
Deamon pong down
Now is 1404873291106
I: Watchdog agent polling time = 1999
```

圖3.10 Watchdog -Backend Demo

Backend 除了需要固定 Ping 對下的 Frontend 是否有回傳 PONG 外，還有 Daemon 的狀態需要檢查。

3.6 功能比較

	Linux Watchdog	Linux Supervisor	My Watchdog
錯誤偵測	系統狀態（CPU、Memory、Temperatures…）、Program 持續寫入	程式狀態（RUNNING、STOPPED…）	PING-PONG 訊息、自定義 Ping 方式
錯誤處置	Reboot System	Restart desire program	Reboot System & Restart desire program
監控層級	Kernel-Space &User-Space	User-Space	User-Space
監控範圍	單機	單機	單機 & 跨網
使用方式	Configure & 檔案 I/O	Script	API
使用難度	難	易	易
使用限制	硬體 & Kernel 支援	程式需跑在前景	能裝 ZeroMQ library & CZMQ Library
使用平台	Linux	Linux	跨平台（以 ZeroMQ 支援為主）

表 3.1 Comparsion between three kind of watchdogs

我們將 Watchdog 難以使用的部分簡化，並且提供可跨網的特性，又因為虛擬機上的系統有可能不同，在實作時選擇 ZeroMQ 同時為了網路及可跨平台的特性。

雙向的 Ping-Pong 機制已經足以發揮監控對方的能力；Reboot 的部分則是通過 configure 設定 Reboot 方式，因此能做到重開系統或者重開程式的分別。

4. 結論

網路本身就是不穩定的通道，但在近期的網路品質不斷地提高下也讓原先電腦一定要有 OS 的概念漸漸改變，各大廠所提供的虛擬系統平台租用服務成了現在的局勢之一，隨著劇增這些系統的使用率就考驗著提供者伺服器的能力，消費者希望盡可能少租用服務換取最大利用率，而系統的可靠程度也會成為他們評斷的標準，不僅是程式的設計功力，就連即使發生錯誤後是否能在最快的速度下回復服務也會對雙方造成很大的影響。

ZeroMQ 針對網路程式中常遇到的問題即使不完美，卻也幫了程式撰寫者不少的忙，不需要考慮同步、連線過程……等問題。在非常忙碌的狀況下也能保持低 CPU 的使用率，程式的重點漂亮的集中於需要做的事而非在網路上的其他問題。

4.1 未來展望

在未來的改進中，本篇於單一 Process 使用了許多結構，在不斷的測試後發現 Memory 很有可能因此不夠，為了能夠讓使用者能更將焦點停留在自己的程式而不是減少不必要的 Memory 消耗上，考慮應該將可能造成耗損的監視 Daemon 部分分別到其他 Process 執行；另一點是需要將此 Watchdog 編成函式庫的形式，讓使用者能夠以 Device 的方式呼叫將會更有利於使用。

相信能跨網的 Wathcdog 在這種雲類型的系統中一定能發揮他應有的功能，他雖然只是系統的一部份，卻能提供系統更強的可靠度，在任何應用上都會成為很大的助力。

5. 參考文獻

[1] dotCloud, “Docker - Build, Ship and Run Any App, Anywhere”, https://www.docker.com/

[2] Pieter Hintjens, “ZeroMQ Messaging for Many Applications”, 03/2013

[3] Addison-Wesley, “Advanced Programming in the UNIX Environment”, 1992.

[4] Supervisor: A Process Control System, “Supervisor Documentation”, http://supervisord.org/index.html

[5] Christian's Blog, “Using the Watchdog Timer in Linux”, http://www.jann.cc/2013/02/02/linux_watchdog.html, February 02, 2013

[6] kunilkuda, “How to Use Linux Watchdog”, http://embeddedfreak.wordpress.com/2010/08/23/howto-use-linux-watchdog/, August 23, 2010

[7] “How To Install and Manage Supervisor on Ubuntu and Debian VPS”, https://www.digitalocean.com/community/tutorials/how-to-install-and-manage-supervisor-on-ubuntu-and-debian-vps

[8] Pieter Hintjens, published by iMatix, “0MQ Code Connected Volume 1” http://hintjens.wdfiles.com/local--files/main:files/cc1pe.pdf

[9] Robertson, Alan. "The evolution of the Linux-HA project." UKUUG LISA/Winter Conference High-Availability and Reliability. 2004.

[10] Soltesz, Stephen, et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.

[11] Abaffy, J. ; Krajcovic, T. “Software support for multiple hardware watchdog timers in the Linux OS”, IEEE, 2010

[12] RADULOVIC, Alex. Private IP communication network architecture. U.S. Patent No 7,215,663, 2007.

[13] NORDMAN, Mikael. Secure access method, and associated apparatus, for accessing a private IP network. U.S. Patent No 6,061,346, 2000.

[14] Addison-Wesley, “Advanced Programming in the UNIX Environment Second edition”, pp. 545-584, 1992

[15] Free Software Foundation, “The GNU Lesser General Public License, version 3.0”, http://opensource.org/licenses/LGPL-3.0

[16] iMatix Corporation, “CZMQ - High-level C Binding for ZeroMQ”, http://czmq.zeromq.org/

[17] RIEHLE, Dirk. Composite design patterns. In: ACM SIGPLAN Notices. ACM, 1997. pp. 218-228.

[18] GAMMA, Erich, et al. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[19] GRAY, Jim; SIEWIOREK, Daniel P. High-availability computer systems. Computer, 1991, 24.9: pp. 39-48.

[20] Hoi Chan and Trieu Chieu. 2012. An approach to high availability for cloud servers with snapshot mechanism. In Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE '12). ACM, New York, NY, USA, Article 6, 6 pages.