中華民國八十四年全國計算機會議論文集
Proceedings of National Computer Symposium 1995

在Linux作業系統中設計與實現容錯設施*
The Design and Implementation of the Fault-Tolerant Facilities in the Linux

楊竹星　林星旭　劉豐榮　曾群偉
C.S. Yang, S.S .Lin, F.J. Lin and C.W. Tseng

國立中山大學資訊工程研究所
Institute of Computer and Information Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C.

摘　要

本系統乃建構在 Linux 的 kernel level 中，利用系統呼叫的方式提供行程容錯的介面，而原有的應用程式並不須要修改原始程式碼或重新鏈結函式庫，即可取得系統所提供的容錯支援。

藉由checkpointing file的製作，保存應用程式的執行影像，有效的降低因硬體錯誤而產生的衝擊，另外，為了減低因容錯服務所帶給系統的額外負擔，應用程式所屬的程式段不放入checkpointing file內，而行程之間的共用資料也只保留一份；當系統發生故障後，回復機制會依據checkpointing file內所記錄的資訊為程式重新取得或重新對應其應有的資源，以保證程式執行的正確性及通透性。

目前本系統所支援的容錯對象為單一工作，工作內允許有多個行程共同運作，但行程之間的 IPC 目前必須以 PIPE, message queue, semaphore 或 share memory 為溝通媒介。

關鍵字：查核點，容錯，Linux 作業系統，回復，轉回。

Abstract

In this paper,we provide two bootstrap-up programs and system function calls to achieve fault-tolerant facilities. When a user program requests the fault-tolerant service,it only needs to invoke the bootstrap-up program and loads the user program itself,or insert the new system function call in source code.

We make checkpointing files to reserve part of the execution image of user programs for the fault-tolerant purpose. To reduce the overhead of checkpoint, the code segment of user program is not written into the checkpointing file. And the shared data between processes are copied into the checkpointing file only once, copy-on-write memory page, share memory, PIPE....

The recovery strategy is based on the information storing in check-pointing files to reget or remap resources of the restarting program. Our strategy will guarantee the user program to run correctness,and users can choose each machine in a homogeneous environment to restart their programs.

The basic unit of this fault-tolerant distributed Linux System is the task. A task may contain several processes,and the communication media between processes can be PIPE,message queue, semaphore and share memory.

keyword : Checkpoint,Fault-Tolerance,Linux, Recovery, Rollback.

1. 簡介

藉由軟硬體的配合，今日電腦世界已不再是一個單打獨鬥的局面，利用網路的連接，電腦之間形成了一個生命共同體，藉由彼此互相支援，而呈現了一股更為強大的計算能力；由於分散式系統具備了資源共享及資源分散管理的特性[1]，所以特別適合容錯系統的發展，當網路中的節點產生硬體錯誤時，可憑藉一些規範與轉換，而在可用的機器上重新啓動某些已失效的程式。

目前有關行程容錯方法的研究，可分為兩個大方向，其一為Checkpoint/Rollback，另一

* This work supported by NSC under grant NO. 84-1123-E-110-008

為N-Modular Redundancy,以下即為此兩種方法的說明:

1.Checkpoint/Rollback[2,3,4,5,6,7,8]

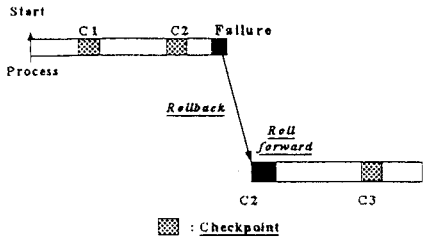

圖 1.1 : Checkpoint/Rollback

此種方法的精神在於週期性的去抓取行程的執行影像，並將所得到的資料做成Checkpointing file,而後將其存入穩定的儲存媒介中如圖1.1中C1,C2及C3,將來如果行程被某些因素干擾而不正常中斷,我們就可以從穩定的儲存媒介中,尋得此行程最近一次所記錄的Checkpointing file,而將它重新啓動,在重新啓動的過程中必需做到將行程的狀態恢復到跟做檢查點時一模一樣,也就是說在檢查點之後至不正常中斷之間,所發生的一些事件(Event)必須利用某些技巧來將它們遮蔽(Mask)掉,我們稱這個動作為Rollback，以保證重新啓動的行程不會因為這些事件的再次發生而誤入歧途,產生錯誤的結果。

Rollback之後，行程將從Checkpoiting file所記錄之處重新啓動。系統必須保證剛剛因為Rollback而被遮蔽掉的事件,必須重新發生一次,這個動作稱為Rollforward,如此程式才能有正確的執行成果。

我們先說明何謂一個一致性的狀態[2](Consistent State),一致性的狀態，視為在系統執行中可能存在的任何一個狀態,包含已存在或有機會存在的任何狀態,而Rollback的目的即為將程式回復到一個一致性的狀態,若程式已處於一致性的狀態,那麼我們就認為程式接下來的所有動作都將會是正確的。

一致性的狀態內不允許有任何Lost messages及Orphan messages發生,如圖1.2所示,若系統在時間點T1時,行程B發生狀況而失效,若系統選擇CA2及CB1為重新執行的依據,則M1就變成了Lost message,因為行程A認為M1已經送過了,而行程B卻認為M1還沒送出來,這種狀況在正常的執行過程中不可能會發生,所以目前系統處於不一致的狀態中。

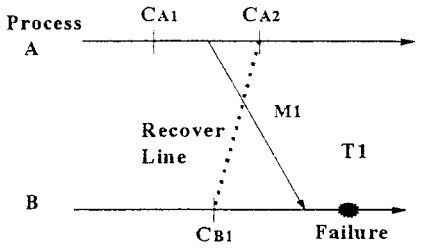

圖 1.2 : Lost Message

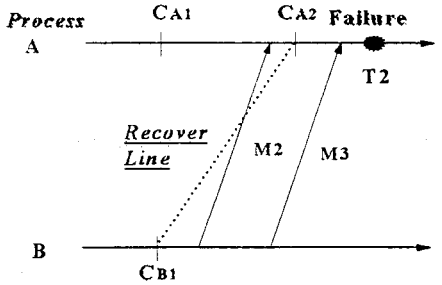

圖 1.3 : Orphan Mseeage

在圖1.3中,行程A在時間T2時發生錯誤,若系統選擇CA2及CB1為重新執行的依據,則M2就變成了Orphan message,因為行程B認為M2還沒送出來,而行程A卻已經收到M2了,這種狀況在正常的執行過程中也是不可能發生的,所以目前系統處於不一致的狀態中。

2.N-Modular Redundancy

這種容錯方法的精神在於利用網路上可用的硬體資源(如圖1.4所示),將所要容錯的行程複製多份並將它們放置在分散式系統中N個可用的節點上,同時執行它們,只要N台機器中有一台能順利執行完此行程,即達成了容錯的目的。

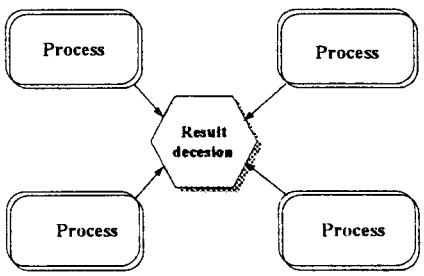

圖 1.4 : N-Modular Redundancy

N-Modular Redundancy的花費相當的高,所以在一般的分散式系統均是採用Checkpoint/rollback的方式來提供行程容錯

的功能，然而在一些對時間敏感度較高的即時系統(Real time system)，N-Modular Redundancy所提供的行程容錯服務則是較佳的選擇。

2. 相關研究

以下將列出相關的三個實作系統，並逐一說明其特色.

1.CONDOR[9,10,11]

CONDOR是由美國威斯康辛大學所發展出來的一套系統，它最大的特色在於使用者無需修改任何已存在的程式，只要將程式重新與CONDOR所提供的特殊C程式庫鏈結，即可以獲得CONDOR有關於Checkpointing file及行程遷徙(Process migration)的服務。

CONDOR完全架構在UNIX的User level上，它製作Checkpointing file的方法如圖2.1所示，藉由core dump產生的core file內含之資料段和堆疊段，以及執行檔內原有的資料，組合而成checkpointing file。

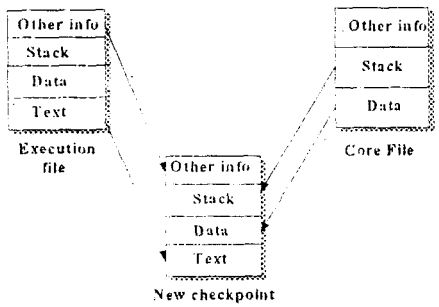

圖 2.1 : CONDOR之checkpointing file的製作方法

2.分散式UNIX系統不中斷執行環境之研究[12]

這個研究計劃的行程容錯部份是以CONDOR的架構為其基本的概念，它也是完全建立在UNIX的User level上，提供了一套特殊的C函式庫，使用者的程式只要與這一個特殊的C函式庫鏈結，即可得到這個系統的服務，它和CONDOR最大的不同點在於Checkpointing file的製作方式，由於CONDOR是從Core file及原執行檔中取得製作Checkpointing file的材料，然後還必須依賴剛製作完成之Checkpointing file內所記錄的資料來重新啓動程式，這樣的做法將牽涉到太多檔案的存取而使系統效能大受影響，同時行程每做一次Checkpointing file就必須重新啓動一次，這也是很不合理的一點。這個研究計劃利用了ptrace以及kvm這兩個系統呼叫來抓取行程的部份執行影像，完成Checkpointing file 的製作，如此一來不僅避免了Core file 的製作，而且行程亦不需要中斷後再被重新啓動，所以系統因執行容錯而產生之負擔應會遠小於CONDOR。

3.CATCH(Compiler-Assisted Techniques for Checkpointing)

CATCH[13]是一個以編譯器(Compiler)技術來達成行程容錯的系統，它已被實現在GNU C Compiler 1.34的修正版中，目前容錯的對象仍限制為單一行程的工作，CATCH藉著程式在編譯的時候插入一些程式片段於可執行檔中，使得這個程式執行時，能夠週期性的建立Checkpointing file。

3. 使用限制

目前我們已經在Linux的Kernel level中完成了有關於單一工作的容錯服務機制，我們將單一工作定位在提出容錯服務要求的行程以及由此行程所衍生出之子行程，也就是這個行程的所有子代(Children)，假設目前系統中有六個行程正在執行(如圖3.1)，其相互的關係是A為B及C的父代(Parent)，B是D與E的父代，而C是F的父代，若只有行程B提出容錯的要求，則系統會將B,D,E看成一個工作，而一起為它們提供容錯服務，A,C,F則被摒除在外。若行程A提出容錯的要求，則系統將對所有行程提供容錯服務。

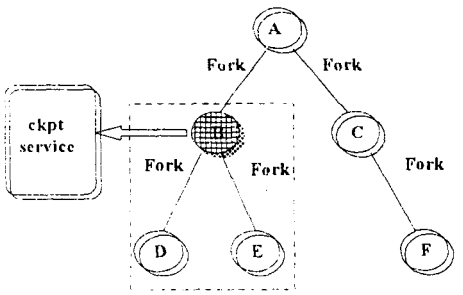

圖 3.1 : Checkpoint服務的對象

以下即為本系統目前所需的執行界面及限制：

檔案系統的根目錄內必須建立一個ckpt的子目錄，並將可靠檔案伺服器上的/ckpt掛上(mount) 區域機器的ckpt，以做為存放ckeckpointing files的地方

若使用者希望不正常中斷的工作能夠在別台機器上重新啓動，則這個工作所有開啓的檔案中不允許有區域(Local)的檔案(標準輸出，輸入，錯誤輸出除外)

.檔案不可同時具有被讀取及寫入的權限，否則不保證其正確性
.Share memory,Semaphore,Message queue及pipe只允許工作內的行程間使用作溝通，否則不保證其正確性
.目前不支援Socket的任何相關動作
.系統中必須有一個ftlinux的帳號，且每台機器上ftlinux的使用者代碼(UID)都必須一樣，使用這個帳號才能使我們的容錯系統動起來

4. Checkpoint

為了要減少製作Checkpointing file花費的時間，我們決定在Checkpointing file內只記錄會隨程式的執行而變動的資料，包括了程式中各行程的資料段，堆疊段以及位於系統核心內的相關資料結構，由於不變的程式段可以從原執行檔中取得，我們並不考慮將程式段放入Checkpointing file內，此外若程式內的某些資料為共享的，則這些資料在Checkpointing file內將只保留一份，減少這些寫入動作，將為系統節省一段相當可觀的時間。

4.1 Checkpointing file的命名規則

在網路上的每部機器都有一個屬於它們自己的名字:Hostname，且Hostname在區域網路(LAN)中是不允許被重複使用的，所以在我們的環境中每個Hostname均是唯一的，利用Hostname加上Command name以及行程代碼(PID)就可以定義出一個區域網路中唯一的Checkpointing file name，利用這個名稱可以很輕易的區別出Checkpointing file與工作間的關係。

我們將為同一個工作儲存兩個不同名稱的Checkpointing file在檔案伺服器中，它們的名稱分別為先前所定義的"Checkpointing file name"及"Checkpointing file name.new"。因為不能保證在製作Checkpointing file的時候，系統絕對不會出任何問題，所以在製作Checkpointing file時先採用"Checkpointing file name.new"這個名稱，而在完成Checkpointing file的製作後，必須執行下列兩件事:

1.若先前已有相同名稱之Checkpointing file存在，則刪除它。
2. 將名稱由"Checkpointing file name.new"改為"Checkpointing file name"。

這種做法能夠保證檔案系統在第一次成功完成Checkpointing file後，無論何時均至少保存一份完整的Checkpointing file，利用下列的判斷即可以得到正確的Checkpointing file(容錯服務啓動之初便會立刻建立一個空的Checkpointing file其名稱即為"Checkpointing file name")

1. 若"Checkpointing file name"與"Checkpointing file name.new"均存在，且"Checkpointing file name"的大小為0，這種情況只存在當系統為某一個程式製作第一個Checkpointing file的過程中不幸夭折時才會存在，由於目前Checkpointing file內的資料並不充足，所以只有重新執行原程式了。

2. 若"Checkpointing file name"與"Checkpointing file name.new"均存在，且"Checkpointing file name"的大小不為0，這種情況只有可能存在於系統為某一個程式製作第二個或第二個以上的Checkpointing file時當機才會存在，因此"Checkpointing file name.new"裡的資料並不是正確的，所以應以"checkpointing file name"內的資訊做為回復機制的輸入。

3. 若只有"Checkpointing file name.new"存在，這種情況為系統在刪除舊有的Checkpointing file後就當掉了，因此還來不及將"Checkpointing file name.new"的名稱更正，由於"Checkpointing file name.new"已包含了有關程式的最新狀態，所以可以利用它來重新啓動程式。

4.2 何時執行checkpoint的動作

當行程由Kernel level要回到User level時，UNIX like的系統會先檢查signal的狀態，之後才將控制權交還給User level的Proess(如圖4.1a所示)，在FTLS中對此一流程做了小小的改變(如圖4.1b所示)，當處理完signal之後，系統會去測試CKPT旗標，以決定是否要為此行程製作Checkpointing file，而後才回User level。

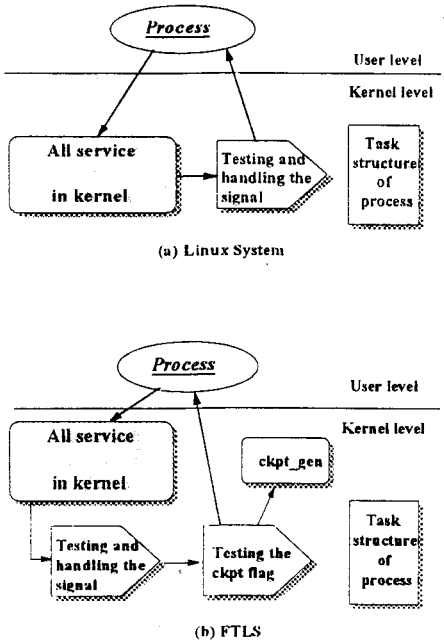

圖 4.1:由kernel level回user level的流程

當CKPT旗標被設定為1時，系統必須先確定工作內所有行程的狀態均為RUNNING，將工作內所有行程的狀態，均設為UNINTERRUPTIBLE，使它們無法繼續執行，並呼叫ckpt_gen為程式製作Checkpointing file，否則必須延後製作Checkpointing file。

5. Recovery

當工作因節點的失效而被迫中斷執行時，網路仍有許多正常的節點可供運用，回復機制可藉由Checkpointing file所提供的資料，而將這些工作重新在別台機器上運行起來。

回復機制所要面臨的困難非常多，以下我們將列舉出幾個較重要的問題，並提出我們因應的對策。

5.1 檔案重新對映

我們在系統中定義了一個NOT_LOCAL旗標，NOT_LOCAL的目的在表示程式被重新啓動時的機器是否與最原先執行這個程式的機器相同，若相同則為0，不同則為1。

在程式被重新啓動的過程中，處理檔案重新對應的政策如下：

1. 若檔案是屬於區域性的檔案且NOT_LOCAL為1，則回復機制將送出錯誤訊息，且立即停止回復的動作。
2. 對於NFS上的檔案，由於Checkpointing file內已保留此檔案位於那一個可靠檔案伺服器中，以及此檔案在這個可靠檔案伺服器內所對應到的Root file handle和本身的file handle，藉由這三個資料結構的運作將可找到原來所曾開啓的檔案，並重新加以引用。

5.2 回復後如何開啓檔案

若程式是在別台機器上重新啓動，在完成回復之後，工作中的某行程要開啓某一個NFS上的檔案，此時問題就出現了，因為目前工作所處的環境已和原來大不相同，所以開檔的動作可能會宣告失敗，如圖5.1所示，在原來的機器MA上，A為NFS1的mount point，而E為NFS2的mount point，在重新啓動的機器MB上，I與J分別為NFS1與NFS2的mount point，若在MB執行Open("/B/E/H",1,0)就會產生錯誤。我們將藉助Checkpointing file所記載MA上的mount table來解決這個問題，如圖5.2所示，當系統接收到開檔的服務請求時，必須先測試NOT_LOCAL的值，若其為1，則表示檔案路徑須要修正。利用MA的mount table，我們可以得知/B/E/H實際是代表NFS2根目錄內的H檔，而NFS2 mount在MB的J目錄上，所以在MB上H的絕對路徑名稱為/J/H，利用這個路徑名稱我們就可以找到正確的檔案。

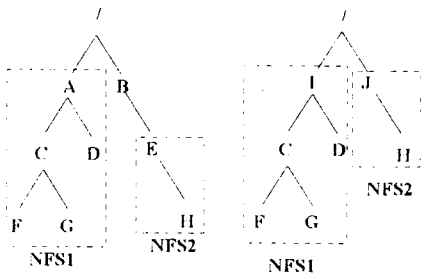

圖 5.1 ： 檔案系統的比較

Mount Point	Mounted File System
/A	NFS1
/B/E	NFS2

Mount Table of MA

Mount Point	Mounted File System
/I	NFS1
/J	NFS2

Mount Table of MB

Pathname /B/E/H ⇨ NFS2/H ⇨ /J/H

圖 5.2 ： 檔案路徑之轉換

5.3 行程間關係的重建與確認

回復後程式內各行程之間的父子關係必須保持和原來的情形一樣，但回復後行程的PID與原來的PID相同的機率並不大，由於有很多函式庫或系統呼叫的操作對象均以PID為準，譬如waitpid，getpid，kill，signal ...，若行程沒有保留舊的PID，則這些動作可能都會停擺或失敗，所以我們在task structure中加入一個O_PID的欄位用以記錄舊的PID，並設立一個新旗標RESTART，當程式被回復，系統就會自動把RESTART設為1。

將上述與PID有關的模組作小幅度的修改，就可以利用O_PID及RESTART的設定與否取得或使用適當的PID值(如圖5.3所示)。

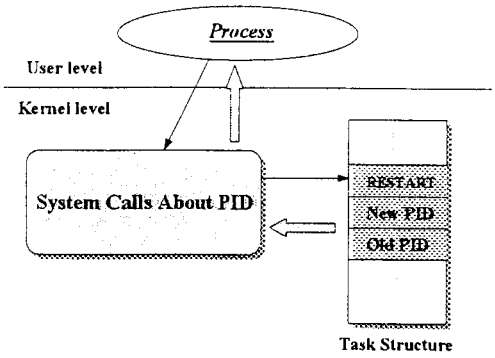

圖 5.3 : 重新啓動後PID的取得方式

5.4 Message queue

行程在Kernel內的相關資料結構，並沒有關於Message queue的資訊，行程對Message queue的認知只有當呼叫msgget時所傳回的MSGID，利用MSGID程式就可以得到系統內某一個message queue的使用權如圖5.4所示製作Checkpointing file的初期，系統會自動搜尋Message queue table這個資料結構，它記錄Message queue的使用情形及訊息存放的位址，並把與目前工作有關的資料，全部寫入Checkpointing file中，重新啓動時將這些資料重新放入Message queue table中沒有被使用到的項目內，利用RESTART旗標的測試(必須修改與Message queue相關的模組)就可正常的操作Message queue，如圖5.5所示。而semaphore及share memory的處理方式與message queue類似。

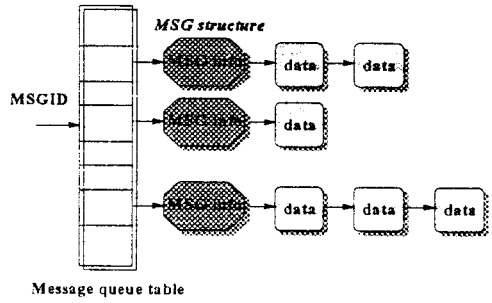

圖 5.4 : Message queue的資料結構

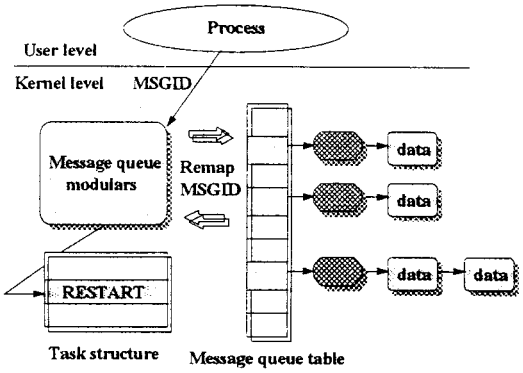

圖 5.5 : 重新啓動後Message queue的操作流程

6. 實驗數據

為了要比較我們的FT-Linux系統與原本不提供容錯設施的Linux系統差異。以及將容錯設施實現在kernel中，是否會比在user level中提供容錯的機制有較佳的成效，所以，我們做了以下的測試。

我們所架設的測試環境如下：

.將一成個人電腦以乙態網路獨立連成一個區域網路，以杜絕任何網路上的干擾。

2.檔案伺服器的組態為

CPU Intel 80486DX-2 66
RAM 8 MB

3.一般節點的組態為

CPU Intel 80486DX-2 66
RAM 12 MB

4.Checkpoint的週期時間為60秒

我們也在Linux的user level中利用ptrace及kmm的部系統執叫抓取行程部分的執行影斷，並且評藉納入 試的項目之一，藉以評斷在kernel level中製作checkpointing file下得為測試資 值得，以下即為測試的資料：

單程式們測試了一個簡單的CPU bound程式，示之及它在原Linux系統(以Linux表示之)及

我們所架構的FT-Linux系統中執行，所得到的數據如下：

SYSTEM	Execution Time
Linux	1518.111 sec
FT-Linux	1518.642 sec

表 6.1 ：測試數據（一）

由表6.1可知FT-Linux並不因增加了容錯服務而使處理一般事務的效率降低，在長達二十分鐘的執行時間下，其所需的執行時間只比Linux多出約0.5秒鐘，overhead幾乎是可以忽略的。

接下來我們以一個Ray tracing的程式作為測試依據，這個程式內包含了複雜的數學運算及大量的檔案寫入動作，所需的執行時間亦相當長，很適合作為我們的test case；另外可藉由參數值的改變而調整其運算的資料量，相對的這個動作亦會影響到Checkpointing file的大小，以下即為測試所得之數據：

kernel ：在kernel level中製作 checkpointing file

user ：在user level中製作 checkpointing file

System	Execution Time	Checkpoint Time per Checkpoint	Overhead
Linux	564 sec	——	——
kernel	652 sec	9.78 sec	15.61%
user	668 sec	11.56 sec	18.44%

The Checkpointing file size is 736K

6.2 ：測試數據（二）

System	Execution Time	Checkpoint Time per Checkpoint	Overhead
Linux	646 sec	——	——
kernel	907 sec	26.1 sec	40.4%
user	954 sec	30.8 sec	47.7%

The Checkpointing file size is 2928K

6.3 ：測試數據（三）

由表6.2及表6.3可以看出，在kernel level製作checkpointing file的速度的確比user level快一些，但是其overhead依然很大，這是因為check-point的週期為60秒，很顯然的，這個週期時間稍嫌短了一些，如果將它增長並配合數據的量測，相信能夠找到一個效能與容錯的平衡點。

7. 結論

當電腦的能力越強，應用範圍越廣時，人們所要承受因電腦當機而產生的麻煩及代價也將隨之升高，所以使電腦系統具有容錯的能力，相信是未來電腦發展的重要指標之一；雖然容錯系統的發展已有一，二十年的歷史，但這方面的技術仍然沒有很成熟，目前仍有很多研究單位致力於容錯系統的探討。

未來我們的發展重點有下列幾項：

1. 使系統能夠自動偵測錯誤的發生，並選定可用的機器作為被回復程式的棲身之處。
2. 配合Load balance的演繹法則，使得網路系統的資源能夠被均衡應用。
3. 將分散式行程容錯的機制實現，提供更強大的容錯能力。
4. 結合Process migration，創造一個更完整的分散式系統。

參考文獻

[1]P.H.Enslow,"What is a Distributed Data Processing System?",Computer,Vol.11,No.1, Jan. 1978.

[2]P.Jalote, "Fault Tolerance in Distributed Systems" , Prentice Hall, Inc,1994.

[3]T.T.Y.Juang and S.Venkatesan ,"Crash Recovery with Littlt Overhead",11th International Conference on Distributed Computing Systems, 1991.

[4]PE.Merlin and B.Randell, "State Restoration in Distributed Systems ",8th International Conference on Fault Tolerant Computing Systems , 1978.

[5]Cheng-Ru Young and Ge-Ming Chiu,"A Crash Recovery Technique in Distributed Computing Systems", Proceedings of the 14th International Conference on Distributed Systems,June,1994.

[6]K.M.Chandy and L.Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems",ACM Transactions on Computer Systems,1985.

[7]R.Koo and S.Toueg , "Checkpointing and Rollback Recovery for Distributed Systems", IEEE Transactions on Software Engineering, Jan 1987.