一個具備動態置換能力的無線感測網路作業系統

林郁傑
國立中興大學資訊科學系
s9456032@cs.nchu.edu.tw

張軒彬
國立中興大學資訊科學系
hpchang@cs.nchu.edu.tw

摘要

在本篇論文中，我們在無線感測網路上支援動態置換的能力，並且實作於 SOS 無線感測網路作業系統之上。透過動態置換的技術，系統能夠隨意的抽換更新模組並且不會造成模組狀態的遺失。我們克服動態置換所需的四項基本條件，並且同時支援應用程式模組和系統核心模組的置換。

除了增加動態置換的功能之外，我們也針對 SOS 的效能做了一些改進。首先，我們改進了 SOS 在配置快閃記憶體空間的機制。接著，我們提升了 SOS 內系統呼叫的呼叫速度。最後，我們將模組連結的動作，卸載到伺服端，而不是在感測節點上執行，如此一來，不僅可以減少節點上記憶體空間的使用，也能節省節點上的電源消耗。

最後，為了評估我們的系統，我們設計了許多應用來驗證動態置換的功能性與正確性，並由實驗結果得知，我們對 SOS 的效能改善可以大幅提昇其執行速度、減少節點上的電源消耗並且延長快閃記憶體的使用壽命。

關鍵詞：感測網路, 動態置換, 動態規劃, SOS

1. 簡介

無線感測網路 (Wireless Sensor Networks) 是由許多具有感測機制和無線通訊能力的微型嵌入式裝置所構成。這些感測節點通常被用來大量佈置於某些區域進行偵測，並週期性的回傳我們感興趣的資料。由於這樣的特性，當日後需要對無線感測網路進行更新，例如除錯、升級版本、或是加入新的功能時，我們無法利用既有的 ISP (In-System Programming) 的技術將所有的感測節點一一回收，然後重新規劃。換言之，我們必須針對無線感測網路的特色，提供一套有彈性且有效率的更新機制。

在本篇論文中，我們在無線感測網路上提供動態置換的能力，並且實作於 SOS (Han, Kumar, Shea, Kohler, & Srivastava, 2005) 之上。相較於一般的網路更新機制，動態置換除了提供系統更新的能力之外，在系統更新完成之後不需重新開機啟動便能直接執行新的程式碼。此外，動態置換也保證在系統更新的過程中，執行狀態能夠被保留下來，並且轉移到新的程式上繼續使用。在實作上，我們不僅支援應用程式層次的動態置換，也同時支援核心層次的動態置換，如此一來，系統便能根據不同的應用型態提供多樣化且富有彈性的系統服務。

除了讓 SOS 支援動態置換之外，我們也同時改進 SOS 的系統效能。首先，我們改進 SOS 配置快閃記憶體時的缺失。在 SOS 中，快閃記憶體的配置方式是採用線性搜尋的方式來尋找可用的記憶體空間，然而，這樣的作法會大大縮短快閃記憶體的使用壽命。我們修改原先的作法以隨機方式動態配置記憶體，平均分散快閃記憶體的讀寫次數，延長其使用壽命。此外，在 SOS 中，當應用程式模組呼叫核心服務時，必須透過 Jump table 以間接方式完成，我們修改為直接的方式，以提升系統服務的呼叫速度。最後，我們將模組連結的動作卸載到伺服端進行。在感測節點上執行模組的連結動作不僅需要花費額外的網路傳輸流量，也需要較大記憶體空間來存取額外的資訊，且在連結的過程中也會進行大量的快閃記憶體存取動作，這些過程都會消耗節點上的電源，縮短節點的存活壽命，因此，我們將模組的連結動作卸載到伺服端進行，以改進上述缺點。

在第二章中，我們會介紹一些應用於無線感測網路上的更新機制，此外，我們也會在第二章中簡單的介紹 SOS 的系統架構和支援動態置換所需要的四個條件。第三章中會介紹我們的系統架構與實作細節。在第四章中，我們會介紹我們所做的實驗與結果。第五章則是結論。

2. 相關研究

2.1 網路更新的種類與方式

目前已有許多的研究針對不同的系統提出相對應的網路更新機制。在本節中，我們將這些網路更新機制進行粗略的分類並加以介紹:

- **Full Image Replacement:**
  此方式較著名的方法有 XNP (Jeong, Kim and, & Broad, 2003) 和 Deluge (Hui, & Culler, 2004)，這兩者都是專為 TinyOS (Hill, et al., 2000; Gay, et al., 2003) 而設計的更新方式。其特色是，每當系統需要更新時，伺服端會將整個新版本的系統核心，驅動程式，和所有的應用程式一次編譯成一個映像檔，然後發佈出去給網路上的各節點。節點收到新的系統映像檔之後便覆蓋舊的映像檔，然後重新啟動

系統，完成整個系統的更新動作。利用這樣的作法，優點在於可以很簡單的完成更新的動作。不過也由於每一次的更新都需要重傳整個系統映像檔，如果更新只是修改小部份的程式碼，這種方式會顯得很沒有效率。

- **Diff-based Approaches:**

  為了改進 full image replacement 方法的缺點，緊接著有學者提出利用 Diff-based 的方式來更新系統 (Jeong, & Culler, 2004; Reijers, & Langendoen, 2003; Dunkels, Finne, Eriksson, & Voigt, 2006)。在 Diff-based 的機制中，首先會比對新舊映像檔之間的差異，並且根據這些差異產生一個更新腳本檔(script)。接著，只需上傳更新腳本和新增加的程式碼到節點上即可。每一個感測節點固定會在 External Flash 中儲存一份系統映像檔的拷貝。當節點收到更新用的資料時，便開始執行所收到的更新腳本。節點會根據腳本的內容，利用儲存於節點內的映像檔複本和所收到的新的程式碼，重新組合成一個新的系統映像檔，並且覆蓋舊有的映像檔，然後重新啟動，執行新的映像檔，如此一來即完成更新動作。

  Diff-based 雖然有效的改進了 full image replacement 的缺點，不過利用這種方式更新系統需要浪費額外的記憶體空間來存放系統映像檔複本，而且更新時也需要花費時間來執行更新腳本以組合新的系統映像檔。

- **Virtual Machine:**

  採用 VM (Levis, & Culler, 2002; Levis, Gay, & Culler, 2005; Liu, Roeder, Walsh, & Barr, 2005) 的方式是在每個節點的作業系統上額外執行一個虛擬機器，然後應用程式皆以虛擬機器所能接受的腳本語言撰寫完成並執行於虛擬機器之上。由於虛擬機器所使用的腳本語言層次比較高，因此程式碼會比較精簡，程式碼行數通常會比較少，這樣有助於減少更新時的傳輸量。此外，每當系統需要更新程式時，只要替換掉執行於虛擬機器之上的應用程式腳本即可，不需要重新啟動系統。

  雖然虛擬機器可以透過極少的傳輸量達到系統的更新效果，不過由於應用程式的執行層次比較高，因此執行的速度會比較慢，且花費的電量也會比較多。況且我們只能對執行於虛擬機器之上的程式碼做更新動作。對於虛擬機器本身及作業系統的部份，都是無法更新的。圖 1 為 Mate (Levis, & Culler., 2002) 的架構示意圖。Mate 是為 TinyOS 系統所設計的虛擬機器，本身以 TinyOS 的元件型式包裝。Mate 藉由與系統內其它元件互動來提供上層應用程式所需要的基本服務。

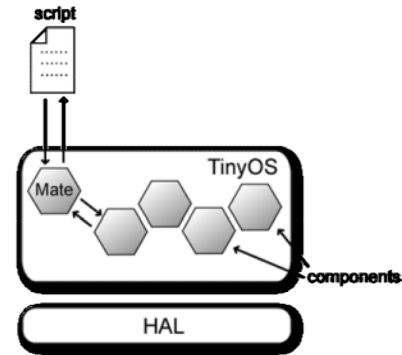

圖 1:Mate 的系統架構示意圖。

- **Loadable Modules**

  採用 Loadable Modules (Dunkels, Gronvall, & Voigt, 2004; Han, et al., 2005) 的更新方式是在每次更新時，只更新有變動的模組。因此傳輸量的大小決定於變動的模組個數。這種更新方式先天條件是必須要有系統的支援才行，系統必須以模組化設計，並且要能支援插入和移除模組才能達到更新的功能。SOS 即是採用這樣的方式，並且只支援應用程式的模組更新。

- **Others:**

  Impala (Liu, & Martonosi, 2003) 本身採用 middleware 的方式來設計。應用程式執行在 middleware 之上。Middleware 由三個部份所組成，分別為 Application Adapter，Application Updater 和 Event Filter。系統利用 Application Adapter 和 Application Updater 來支援更新和調整應用程式。然而事實上，這個系統並未實作在無線感測網路之上。

  MiLAN (Heinzelman, Murphy, Carvalho, & Carvalho, 2004) 會不斷的蒐集許多資訊，其中包含應用程式所需要的 QoS 等級、各應用程式的重要性、和整個網路所擁有的資源及頻寬等，綜合這些資訊，MiLAN 會對網路進行調整，以求盡量滿足每個應用程式的需求。雖然 MiLAN 可以動態的改變應用程式的行為，但是不支援程式碼的更新。

2.2 SOS 系統架構

SOS (Han, et al., 2005) 是專為感測器網路所設計的作業系統。系統本身以模組化設計(Dunkels, Gronvall, & Voigt, 2004)，支援應用程式模組的插入和移除。系統的運作由一個基本的核心，加上數個應用程式模組間相互合作完成工作。如圖 2 所示，模組與模組之間的溝通可以透過直接的函式呼叫或是利用訊息的方式進行。每一個模組會在檔頭中註明該模組有提供那些函式給其它模組使用，或是需要使用到那些由其它模組所提供的函式。當模組

插入到系統時，系統會檢查這些資訊以協助模組間的連結動作。連結完成後的模組便可以對其它的模組進行直接的函式呼叫。此外，如果模組需要對核心的函式進行呼叫時,則需透過核心內的 jump table 來完成。Jump table 裡記錄了核心內提供服務的函式位置。模組必須透過 jump table 以間接的方式對核心函式進行呼叫。

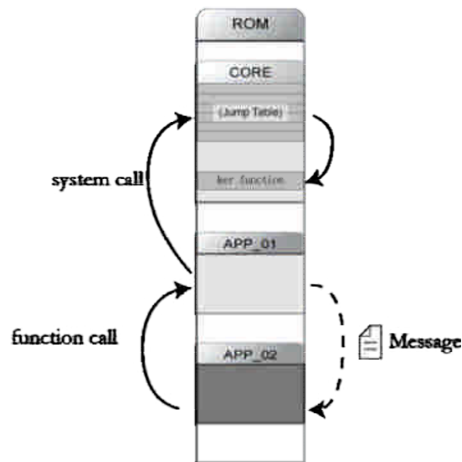

圖 2：SOS 模組之間的溝通可以透過函式呼叫的方式或是訊息的方式來進行。模組進行核心函式呼叫時則需透過 jump table 來進行。

圖 3 顯示 SOS 內部排程的方法，共包含三個佇列，每個佇列各有不同的優先權，分別為 high、system、和 low。SOS 會根據訊息的優先權分別將訊息置入相對應的佇列中。接著，SOS 會從最高優先權的佇列開始尋找訊息，並且以 FIFO 的方式一次處理一個訊息。然後，系統會根據訊息內的資訊，將訊息丟到對應的模組。每個模組內都有著自己的訊息處理函式，當收到訊息時，模組會根據訊息的型態做出相對應的動作。

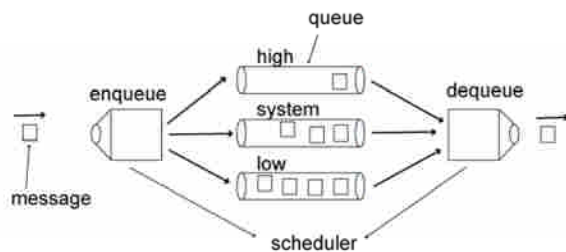

圖 3：SOS 的排程方式。enqueue 會根據訊息的優先權，將訊息丟到對應的佇列中。dequeue 會依照佇列的優先權，以 FIFO 的方式，依序從中取出訊息並且將訊息丟給對應的模組處理

2.3 動態更新的條件

根據 K42 (Soules, et al., 2003) 的作法，系統欲達到動態置換的能力必須實現底下四個條件:

1. Component Boundaries:

一個定義明確的模組可將相關的資料和函式封裝在模組內，並且盡可能的降低模組和模組之間的相依性。藉由清楚的定義模組，當置換發生時，系統才能正確決定應該置換的程式碼與變數有哪些。一般來說，模組的定義可以藉由物件導向語言的協助，或是自身系統的設計達成。

2. Component State Transfer:

在動態置換的過程中，舊有模組的執行狀態必須被轉移到新模組上繼續使用，不應因為模組的更新而遺失，以免造成模組上重要資訊的遺失，影響系統執行的正確性。

3. Mutually Consistent States:

當模組欲進行動態置換時，必須確保模組在置換前後是處於“一致”的狀態。所謂一致的狀態是指模組在置換的過程中，不會有其它因素影響到模組內部狀態的穩定性，造成模組內部資料變的不可預測。這個問題類似傳統作業系統的 race conditon。最簡單的解決方法就是確保模組在置換過程中是靜止的，除了置換程式，沒有其它程式會呼叫或是使用到將被置換的模組。例如 K42 便是利用 generation count 的方式，來判斷模組是否處於靜止狀態(Soules, et al., 2003)。其它的作法有 reader-writer lock (Soules, et al., 2003)或是 stack trace (Lee, C& hang, 2006) 等。

4. External references:

當舊模組被置換後，新模組所使用的記憶體位置不一定會和舊模組的相同。因此，模組所提供的函式位置也會不相同，如此一來，外來的呼叫會找不到新的函式位置。因此，一個支援動態置換的系統必須提供適當的機制來重新連結這些呼叫到新模組上的函示。

3. 設計與實作

3.1 系統實作

在我們的實作中，系統可以對應用程式模組或是核心模組進行動態置換。而在進行動態置換時，無論是對應用程式模組，或是對核心模組置換時，所需要的基本步驟如下:

◆ **配置記憶體空間：**

在新模組上傳到節點之前，必須先在節點的記憶體中找到一塊適當大小的空間來存放將要上傳的新模組。

◆ **上傳新的模組：**

節點的更新動作是靠著SOS內部定義的更新協定來完成，但是該協定只支援插入模組和移除模組的功能。我們稍微修改原有 SOS 的模組更新協定，使其支援模組置換的功能。

◆ **進行狀態轉移：**

在舊模組被移除之前，我們讓舊模組有機會可以進行狀態的轉移。舊模組可以將自身內部的執行狀態適當包裝，利用訊息的方式轉移給新模組繼續使用。

◆ **重新連結函式呼叫：**

由於新模組所處的記憶體位置和舊模組不同，因此新模組內所提供的函式位置也會與舊的函示位置不同。如果有其它的模組參考到被置換的模組時，那麼這些參考位置必須被修正，導向新的位置。

◆ **移除舊模組：**

當舊模組完成狀態轉移動作之後，便不可再進行任何的動作，否則會造成模組狀態的再次改變。因此，在狀態轉移之後，系統會立即將舊模組從系統內移除，並且適放其所占有的記憶體空間。

◆ **更新 Jump Table (核心模組才需要做)** ：

在 SOS 中，應用程式模組存取核心資源時，需要透過 jump table。因此，如果被置換的核心模組本身有輸出函示到 jump table，那麼在置換的過程中，我們必須同時修正 jump table 上的值。

在接下來的部份，我們會針對動態置換所需要的四項條件，分別從應用程式和系統核心兩個方向切入，並解釋我們的作法。

3.1.1 Component Boundaries:

應用程式部份:

由於 SOS 本身是模組化設計，因此我們直接採用 SOS 的設計。在 SOS 中，當撰寫模組時，必須先定義模組的檔頭。模組檔頭內包含關於此一模組的一些重要訊息，例如模組內提供給外界(Export)呼叫的函式記憶體位置、識別碼…等。採用 SOS 原本的模組包裝方式，可以讓既有的模組不用重新撰寫包裝也能相容於我們的系統中。

核心部份:

由於 SOS 本身不支援核心程式碼的置換，所以要讓核心程式碼能夠支援動態置換，必須對系統核心程式碼進行模組化的工作。基本上，我們將要被置換的核心程式碼採用和應用程式同樣的模組包裝方式，並且修改核心內部的函式呼叫方式。減少核心模組彼此之間的相依性，以方便模組置換工作。

3.1.2 Component State Transfer:

應用程式部份:

欲實現狀態轉移，首先，我們要求每一個模組新增一個訊息處理函式(message handler)，處理我們新定義的 MSG_SWAP_PACK 訊息。

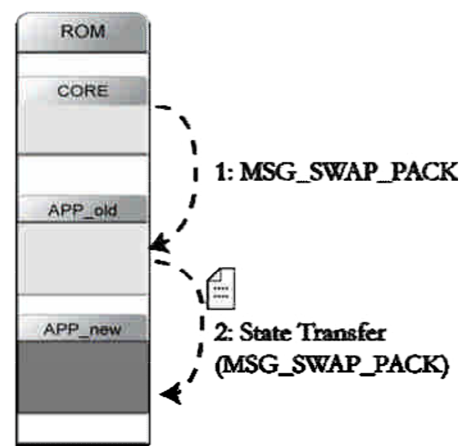

圖 4：狀態轉換的過程。

如圖 4 所示，每當舊模組被置換前，系統會發出一個 MSG_SWAP_PACK 訊息給舊模組。收到該訊息的舊模組必須將自己的狀態利用訊息的方式包裝在 MSG_SWAP_UNPCK 訊息裡，然後傳送給新模組。新模組則必須要處理由舊模組傳過來的 MSG_SWAP_UNPACK 訊息，並且將其中的資料解開，回覆模組之前的狀態。

核心部份:

由於核心模組的包裝方式和應用程式並無差異，因此在狀態的轉換上，我們採用和應用程式相同的方式來轉換核心內部的狀態。

利用訊息進行狀態轉移時，我們可能會遭遇到一個問題。如圖 5 所示，假設在佇列中的三個訊息 A、B、和 S 都是要送給新的模組，且 S 訊息是舊模組送給新模組的狀態轉移訊息，那麼我們會發現，在新模組收到 S 訊息之前，A 訊息和 B 訊息會先送到新模組中，這樣新模組便會在還沒進行狀態回覆的情況下處理新訊息。為了解決這個問題，我

們在S訊息進入排程之前先檢查佇列中是否有正要送給尚未進行狀態轉移的模組(例如：A 和 B)，如果有的話，便將狀態轉移用訊息(S)的優先權提升到最高，讓模組先行完成狀態轉移的工作，然後再處理陸續到來的其它訊息。

除了模組的狀態轉移之外，我們的系統也會進行系統資源的轉移動作。舊模組在執行過程中所註冊使用的系統資源，如 timer 等，這些資源在模組進行動態置換時，我們的系統會一併將這些資源轉換到新模組上，讓新模組能夠直接使用，省去重新註冊並初使化系統資源的時間。

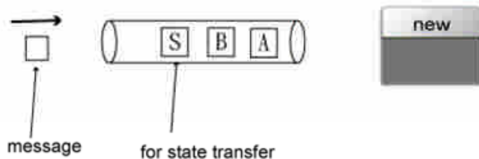

圖 5：狀態轉移時可能的問題。A 訊息和 B 訊息會搶在狀態轉移訊息 S 之前先送到新模組上。

3.1.3 External references:

應用程式部份:

在 SOS 中，每當模組使用到其它模組所提供的函式時，模組會事先留下待連結的指標，並且告知 SOS 有多少個指標需要被連結。SOS 會找到這些空指標，並且依照模組的要求一一將指標連結到正確的函式記憶體位置。因此，模組便能透過函式指標呼叫到其它模組所提供的服務。

但是，當模組被置換時，模組之間的連結可能會變得無效。因此在新模組置換舊模組時，我們讓 SOS 再做一次模組之間的連結動作。例如圖 6 中，APP_01 原本是使用 APP_02_old 內的服務，此時有個新的 APP_02_new 被安裝到系統中，那麼我們會將 APP_01 重新連結到 APP_02_new 中。

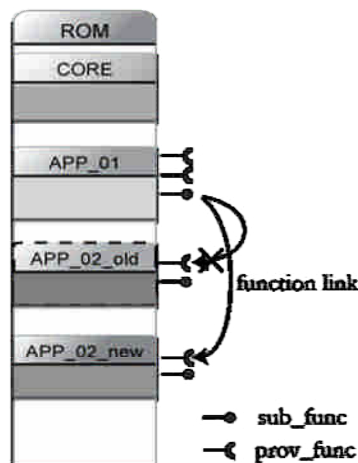

圖 6：模組進行置換後，需要對函式呼叫進行重新連結。

核心部份：我們分兩部份探討：

(1)：解決核心模組與核心模組之間的參考:

為了減少核心模組之間的相依性，我們將核心模組之間的關連以函式指標隔開。核心模組如果想使用核心內其它成員的函式時，則必須透過函式指標來進行。相對的，核心模組也能透過動態更改函式指標的值，來修正所提供服務的函式位置。例如：我們想置換核心函式 mq_enqueue，其原本的呼叫方式為

mq_enqueue(&schedpq, m);

我們要將它改寫成

(*mq_enqueuePtr)(&schedpq, m);

也就是透過函式指標的方式來呼叫核心函式。如圖 7 所示，ker_module_01 模組利用函式指標來使用 ker_module_02 模組內的資源。這樣的作法被廣泛使用在 Linux 的核心程式碼中。透過函式指標能夠降低模組之間的相依性。每當有新的核心模組被安裝時，新模組只要修改核心內函式指標的值，即可將呼叫轉移到新的函式。

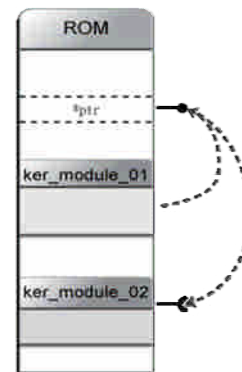

圖 7：核心模組間連結的方式。

(2)：解決由 application 模組到 kernel 模組的參考:

核心內的 jump table 會記錄核心函式的記憶體位置，並且提供給應用程式模組在呼叫時查詢使用。如果提供服務的核心函式因為動態置換的關係而更換了記憶體位置，並且該函式又有輸出到 jump table 上的話，那麼 jump table 內的值也必須做對應的修正才行。

3.1.4 Mutually Consistent States:

要進行動態置換之前，我們必須先確定將被置換的模組，其上所有的服務都已結束。避免動態轉換的動作照成系統內的狀態不穩定。由於 SOS 採用 FIFO (First-In-First-Out)的方式排程，同時間一次只會有一個訊息被處理，而且只有在模組完成此一

訊息的處理之後，下一個訊息才會被處理。因此，在動態置換的過程中，同時間內不會有其它的模組被執行而影響到系統的正確性，因此我們不必特別針對此一問題設計解決方案。

3.2 效能改進

在本論文中，我們除了在 SOS 上實現動態置換應用程式模組與系統核心模組的功能，也同時針對 SOS 的一些缺點，改進其效能。在本節中，我們將一一介紹我們的改進方法。

3.2.1 卸載核心模組的連結動作到伺服端

以模組方式的機制更新系統時,都必須在節點上對模組進行連結的動作，這樣模組才能正常運作。這是因為當模組使用到核心內部的資源時，由於在編譯的過程中，編譯器並無法確切知道這些核心資源的位置，因而只能留下一些資訊，等待接下來的的連結器完成連結動作。但是，為了要對模組進行連結, 下載的模組必須夾帶其它額外的資訊，例如：symbol table 和 relocation table 等等。然而這些額外的資訊會增加模組下載時的傳輸量，而且在節點上執行連結動作也會花費額外的電力(Marr′on, et al, 2006) 。

因此，我們實作一個 patch_tool，用來協助編譯器，讓核心模組可以直接在主機上完成連結的動作。首先，我們會在伺服端留下目前各節點正在使用的系統映像檔複本,透過系統映像檔內的 symbol table，我們可以查到所有核心內部資源的位置。patch_tool 會利用這個映像檔複本將模組內編譯器無法解析的部份填上正確的值，然後再送交給編譯器完成正確的編譯。如此一來，編譯好的核心模組只要在伺服器端連結一次就好，便能夠直接和核心資源的記憶體位置連行連結。圖 8 顯示整個執行的過程。

利用這樣的方法不僅可以加快整個感測網路上模組更新的速度，也省去感測節點執行連結動作所需要的計算工作和龐大的記憶體讀寫次數，這代表我們節省了節點所需耗費的電力，讓節點存活的時間更長久。

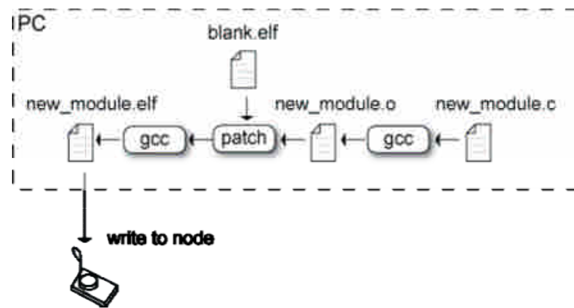

圖 8：核心模組的編譯流程

3.2.2 改良系統呼叫的速度

在 SOS 中，每當應用程式要呼叫核心函式所提供的服務時，都必須透過 jump table 查詢核心函式所在的記憶體位置，然後才能跳到該函式的位置執行動作。因為大部分的應用程式都必須依賴系統所提供的服務，因此，核心函式會經常被呼叫，而每次呼叫都必須透過 jump table 來完成，長久下來，累積的負擔(overhead)將非常可觀，因此，jump table 的實作方式並不是很有效率。

我們想加快應用程式模組對核心函式的呼叫速度。如圖 9 所示，我們修改 SOS，讓應用程式模組在查尋過第一次 jump table 之後，就將所查到的位址記錄下來。之後的呼叫便直接利用之前查尋所記錄下來的快取進行呼叫，省去接下來每次要呼叫同一個函式所需要的表格查尋時間。

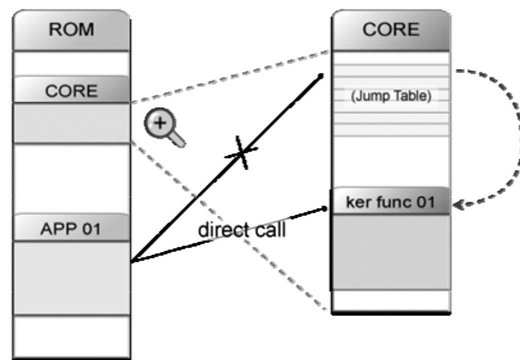

圖 9：改良系統呼叫。

然而採用快取的方式加快呼叫方式可能會遭遇到一個問題。當核心模組被置換之後，舊有的核心函式位置也會跟著變更，因此之前應用程式模組透過 jump table 所快取的值也會變得無效。為了解決這個問題，我們讓系統在核心模組置換之後主動廣播一個訊息給所有的應用程式模組，告知應用程式模組應該重新快取 jump table 內的值，如此一來應用程式模組便能重新呼叫正確的核心函式。

3.2.3 改良系統配置 Flash 記憶體時的缺陷

在 SOS 中，每當要安裝新的模組時，系統必須幫模組在 Flash 記憶體上配置一塊空間。SOS 為了方便配置的工作，會事先將 Flash 記憶體切割成好幾個固定大小的區塊，當需要動態配置記憶體時，SOS 會尋找一塊空的記憶體區塊配置給該模組。在尋找可用區塊時，SOS 是採用線性搜尋的方式。但是，如果模組的更動很頻繁，那麼線性搜尋的方式可能會造成前幾個區塊的使用率偏高。由於 Flash 區塊的讀寫次數是有限制的，因此 SOS 原本的作法並不是很理想。

為了改善這種情形，也就是達到 wear-leveling 的目標，我們將 SOS 的線性搜尋方式修改為隨機尋

找的方式，以期讓 Flash 記憶體中每一個區塊的使用率平均。

4. 實驗與結果

4.1 實驗環境

我們實作的硬體平台為 Mica2。Mica2 是 Crossbow 公司專為無線感測網路所設計的感測節點，其中整合了微處理器(Atmega128)、Flash ROM (512 Kbytes)、無線裝置…等等，並可外接感測器(MTS310)。Mica2 需配合 programming board 使用，才能將檔案從 PC 端透過序列埠經由 programming board 寫入到感測節點上。我們所使用的 programming board 為 Mib510。軟體部份我們使用由 UCLA 發展的無線感測網路作業系統 SOS 1.7 版。

4.2 實驗結果

4.2.1 核心大小比較

表一顯示我們的系統與 SOS 的核心大小比較。由表 1 可以得知支援動態置換的系統只比原來的 SOS 核心多出約 1 Kbytes 左右的記憶體空間。

表 1：核心大小比較(單位: bytes)

	kernel	
	Program	**Data**
Original Version	38940	2829
Modified Version	40056	2839

4.2.2 安裝模組所需花費的時間比較

表 2 和表 3 顯示動態置換和單插入一個 null 應用程式模組與核心模組兩者之間的時間花費比較表(不包含狀態轉移的時間)，null 模組是一個空的模組，並不做任何的事情。動態置換在最後需進行移除舊模組的動作，這是單插入一個模組所沒有的，因此整體的時間花費會比單插入一個模組高一些。在表 3 中，如果插入的核心模組內有函式必須輸出到 jump table 時，那麼系統會在置換的過程中修改 jump table 的值，由於修正 jump table 必須執行快閃記憶體的寫入動作，因此會花費比較多的時間。

此外，我們針對不同大小的模組分別測試置換模組的時間花費。圖 10 中，我們置換三個應用程式模組，分別為 null 模組(20 bytes)，surge 模組(688 bytes)，和 tree_routing 模組(2608 bytes)。圖 11 中，我們測試三種我們所實作的核心模組，分別為變更核心排程用的模組 scheduler，變更核心快閃記憶體配置用的 flash 模組和一個不做任何事情的 ker_null 模組。

表 2：安裝應用程式模組所需花費的時間比較(單位: clock cycles)

	Application module	
	Insert	**Hot-swap**
Check module list	17	15
Register new module	47	45
Remove old module		53

表 3：安裝核心模組所需花費的時間比較(單位: clock cycles)

	Kernel module
	Hot-swap
Check module list	13
Register new module	46
Remove old module	54
Revise Jump Table	1202

在模組註冊的過程中，我們會將系統原本用來管理舊模組時所使用的資料結構直接轉移給新模組使用，這樣的作法可以省去資料結構在釋放並且重新初使化所需花費的時間，因此在動態置換的過程中，註冊模組的時間花費會比原來單純插入模組時為少。

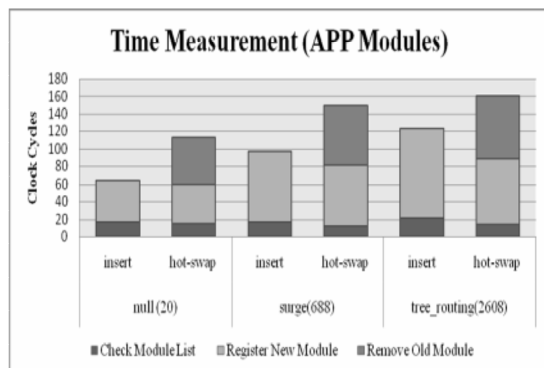

圖 10：三個不同大小的應用程式模組在以插入和動態置換兩個動作上所需花費的時間比較。

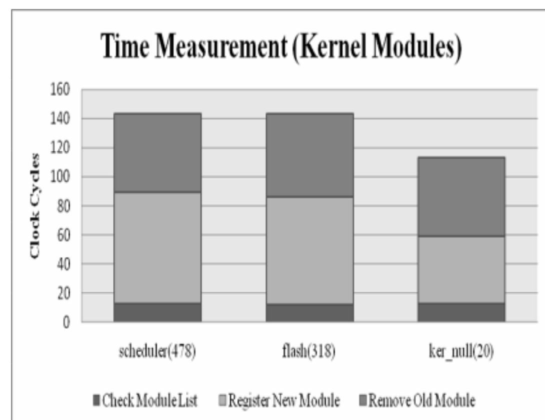

圖 11：三個不同大小的核心模組在以插入和動態置換兩個動作上所需花費的時間比較。

4.2.3 狀態轉移所必須花費的時間

我們將整個狀態轉移過程分成三個步驟，分別測量其需要花費的時間。首先是舊模組將狀態包裝成訊息所需的時間，接下來是訊息的傳送時間，最後是新模組收到訊息、取出訊息內的資料、然後將狀態回存所需的時間。表 4 顯示實驗的結果。在步驟二中，由於訊息的傳遞必須經過在訊息佇列中等待排程，因此會耗費較多的時間。

表 4：狀態轉移所需花費的時間(單位: clock cycles)

Pack		
min	avg	max
6	6.6	7
Transfer		
min	avg	max
196	199.7	203
Unpack		
min	avg	max
5	6.1	9

4.2.4 呼叫系統服務所需要的時間

表 5 顯示呼叫核心函示所需的時間。由表 5 可知，將查尋 Jump table 的結果快取起來，並在以後呼叫時直接利用快取的值進行系統呼叫所需花費的時間比原始的 SOS 所採用的系統呼叫方式可節省約 16%的時間。

表 5：呼叫核心函式所需花費的平均時間

Original Version	Modified Version
1.63	1.362

4.2.5 置換核心排程模組的實驗結果

在這個實驗中，我們修改 SOS，並利用動態置換的方式來置換核心內的排程模組。我們將優先權為 low 的訊息佇列再分成兩個等級，分別為 low-1 和 low-2 兩個等級，且 low-2 的優先權大於 low-1，藉此來增加系統內訊息優先權的選擇。

如圖 12 所示，為了驗證核心模組被正確的置換，在本實驗中，我們安排兩個應用程式模組 APP01 和 APP02，其中 APP01 這個模組會固定每秒發出二個訊息 A 和 B，且 A 和 B 的優先權分別為 low-1 和 low-2。APP02 模組則會接收由 APP01 傳來的訊息，並且記錄所接收到的訊息順序。我們在實驗進行到一半的時候置換我們所實作的核心模組，並且觀察 APP02 接收訊息的順序變化來驗證核心模組是否被置換成功。

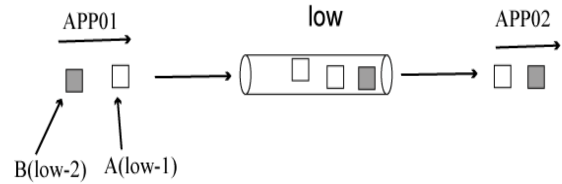

圖 12：置換核心排程模組的示意圖。A 訊息雖然先送，不過由於在新模組中的優先權較低，所以會比較晚送到 APP02。

表 6 列出 APP02 所接收到六次訊息的時間。由於一開始發送的訊息順序是先 A 後 B。因此由前三次可以看出 A 比 B 早收到。但是從第三次之後，我們置換了核心的排程模組，新的模組會區別 low-1 和 low-2 兩個優先權的訊息，且由於 low-2 的優先權大於 low-1 的關系，因此雖然 APP01 發送訊息的順序是先 A 後 B，不過 APP02 接收訊息的順序則變成先 B 後 A。

表 6：訊息接收的時間順序(單位: clock cycles)

	1	2	3
A	3809928	4040384	4270846
B	3809982	4040438	4270903
動態置換			
	4	5	6
A	4501364	4731809	4962267
B	4501248	4731703	4962161

4.2.6 置換核心快閃記憶體空間配置模組的實驗結果

我們利用動態置換的方式來置換系統中配置快閃記憶體的核心模組，將原本 SOS 線性搜尋方式改為隨機搜尋。在本實驗中，我們分別準備不同個數的應用程式模組，並且執行 1000 次一連串動作，以驗證我們的改良情況。模組可以進行的動作有插入，移除，和動態置換三種，而且模組的挑選與要進行的動作一切都是隨機決定的。其中，圖 13 表示從 5 個模組中，隨機挑選模組進行隨機動作的結果。圖 14 和圖 15 中則顯示當模組的個數為 10 和 15 時的實驗情形。由圖 13、圖 14、和圖 15 可以很明顯的看出，採用原始 SOS 的線性配置機制，快閃記憶體內的前段區塊使用率會很吃重，這樣會快速縮短快閃記憶體的使用壽命。而我們修改過後的版本採用隨機配置的方式進行，記憶體的使用率變的很平均，也就是快閃記憶體內的每一個區塊都能被平均的被使用到。

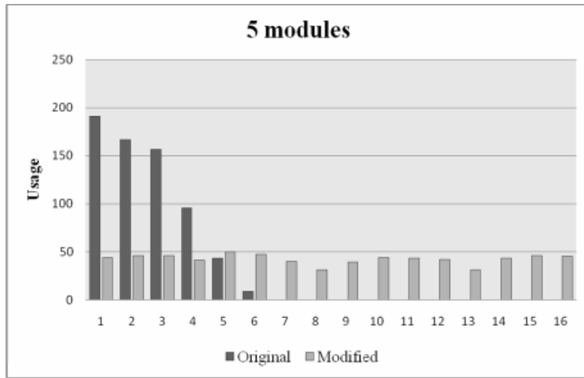

圖 13:使用五個模組時，比較以線性的方式和隨機配置方式之間效率的不同。

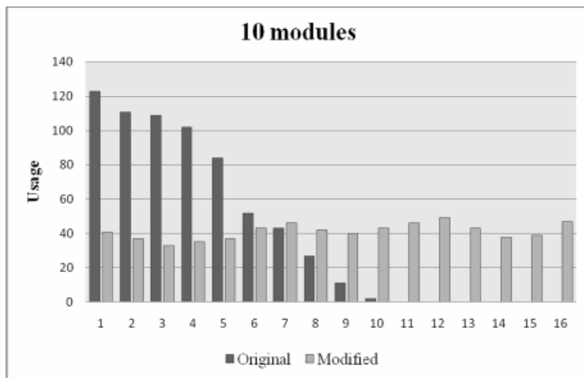

圖 14：使用十個模組時，比較以線性的方式和隨機配置方式之間效率的不同。

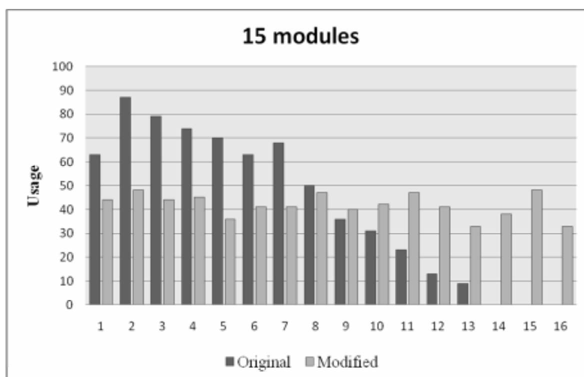

圖 15：使用十五個模組時，比較以線性的方式和隨機配置方式之間效率的不同。

4.2.7 模組狀態轉移實驗

在這個實驗中，我們利用 surge 此一應用程式模組來驗證我們狀態轉移的正確性。Surge 應用程式模組每隔一段時間便會偵測周遭環境的光度，並且將其值(value)回傳，而每一個回傳的封包都會被標記一個流水號(seq_no)。我們修改 surge 應用程式模組，新增一個內部的狀態變數 average，記錄每一次讀取到的值的平均值。我們利用動態置換的方式置換原有的 surge 模組，以我們新的模組取代之。

表 7 顯示實驗前後 surge 模組內部各個變數的變化情形。由表 7 可知，seq_no 有從舊模組最後的值銜接下來，這證明了舊模組內狀態有成功被轉換到新模組上。此外，在動態置換進行之前，average 這個變數是不存在的，因此沒有數值。然而在進行動態置換之後，我們可以發現 average 開始有數值，也驗證了我們的系統可以在原本的模組狀態中新增新的狀態變數。

表 7：動態置換後模組內狀態的變化

	1	2	3	4	5
Value	0x21C	0x237	0x23F	0x21A	0x240
Seq_no	0x5	0x6	0x7	0x8	0x9
Average	N/A	N/A	N/A	N/A	N/A
		6	7	8	9
Value	動態置換	0x213	0x22A	0x244	0x211
Seq_no		0xA	0xB	0xC	0xD
Average		0x213	0x21E	0x231	0x221

5. 結論

動態置換在無線感測網路的開發及維護上，提供了一種極度彈性的方式。我們藉由修改 SOS，在 SOS 上實作動態置換的功能，讓 SOS 能夠根據需求，自由地變更置換舊有系統上的應用程式模組，或是核心模組。此外，動態置換也能確保在模組更新的過程中，系統不需要重新啟動，而舊有模組的執行狀態能夠被轉移到新的模組上繼續執行工作。

此外，我們也改善 SOS 的效能，如提升系統呼叫的速度、平均使用快閃記憶體的所有區塊、將核心模組的連結動作卸載到伺服端上等。由實驗結果顯示，這些機制能夠降低系統的負擔，提升了系統整體之效能。

致謝

本論文承蒙國科會計畫編號 NSC 95-2221-E-005-028-的經費補助，研究方能順利進行，特此致謝。

參考文獻

[1] A. Dunkels, B. Gronvall, and T. Voigt (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In First IEEE Workshop on Embedded Networked Sensors. 455-462.

[2] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt (2006). Run-time dynamic linking for reprogramming wireless sensor networks. In Proceedings of the ACM 4th international conference on Embedded networked sensor systems (Sensys), pages 15–28.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E.

Brewer, and D. Culler (2003). The nesc language: A holistic approach to networked embedded systems. In ACM SIGPLAN Conference on Programming Language Design and Implementation.

[4] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava (2005). A dynamic operating system for sensor nodes. In Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys), 163–176.

[5] W. Heinzelman, A. Murphy, H. Carvalho, and M. Perillo (2004). Middleware to support sensor network applications. IEEE Network 18(2004) 6–14.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister (2000). System architecture directions for networked sensors. In Architectural Support for Programming Languages and Operating Systems, 93–104.

[7] Jonathan W. Hui and David Culler (2004). The dynamic behavior of a data dissemination protocol for network programming at scale. In Proceedings of the 2nd international conference on Embedded networked sensor systems, 81–94.

[8] J. Jeong, S. Kim, and A. Broad (2003). Network reprogramming, tinyos documentation.

[9] Jaein Jeong and David Culler (2004). Incremental network programming for wireless sensors. IEEE SECON, 25-33.

[10] Joel Koshy and Raju Pandey (2005). Vm*: Synthesizing scalable runtime environments for sensor networks, In Proceedings of the third international Conference on Embedded Networked Sensor Systems (Sensys), 189-198.

[11] Yueh-Feng Lee and Ruei-Chuan Chang (2006). Hotswapping linux kernel modules. J. Syst. Softw., 79(2):163–175.

[12] P. Levis and D. Culler (2002). Mate: A tiny virtual machine for sensor networks. In International Conference on Architectural Support for Programming Languages and Operating Systems, 85-95.

[13] P. Levis, D. Gay, and D. Culler (2005). Active sensor networks. In Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI), 100-110.

[14] Hongzhou Liu, Tom Roeder, Kevin Walsh, and Emin Gun Sirer Rimon Barr (2005). Design and implementation of a single system image operating system for ad hoc networks, In Proceedings of The International Conference on Mobile Systems, Applications, and Services (MobiSys), 149–162

[15] T. Liu and M. Martonosi (2003). Impala: A middleware system for managing autonomic, parallel sensor systems. In ACM SIGPLAN Symp,. 107 - 118.

[16] Pedro Jos′e Marr′on, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel (2006). Flexcup: A flexible and efficient code update mechanism for sensor networks. In Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN), 212–227.

[17] N. Reijers and K. Langendoen (2003). Efficient code distribution in wireless sensor networks. In Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), 60-67.

[18] Craig A. et. al., (2003). System support for online reconfiguration. In Proc. of the Usenix Technical Conference, 141-154.