以網路模擬軟體 ns 比較 TCP 之版本及啓始視窗大小

蔡昌憲 林盈達

投稿領域： 其他-效能分析模擬

國立交通大學資訊科學系

新竹市大學路 1001 號

TEL: (03)5712121 EXT. 56667

FAX:(03)5712121 EXT. 59263

EMAIL: gis89524@cis.nctu.edu.tw ydlin@cis.nctu.edu.tw

主要聯絡人： 蔡昌憲

摘要

本文首先介紹一開放原始碼的網路模擬軟體-ns，ns 目前是 VINT 計畫的一部份，不僅免費、更有豐富的內建模組，所以愈來愈多研究機構採用 ns 來進行網路模擬。本文介紹將 ns 的架構、提供的模組、以及如何使用 ns 進行模擬。並提供兩個實例，第一個實例比較各種 TCP 實作版本的效能，一一驗證其運作的過程與效果。第二個實例示範如何在 ns 中加入新的功能，並比較各種不同 initial window size 對 TCP 效能的影響，發現在封包漏失較少時，增大 initial window size 對效能有正面的影響。

關鍵字: network simulator，TCP flow control，initial window size。

1. 動機與歷史

隨著網際網路的快速成長，發展出許多新的網路協定。在早期，當新的演算法或協定設計完成時，研究人員多藉由實驗或是以數學模型的方式，來驗證其正確性或效能。但是現今的網路環境十分複雜，建構一個新的實驗環境相當昂貴，而且這個環境很可能不能應用在下個實驗中，更不能分享給全世界其他人使用；而以數學分析的方式常常因複雜度過高而難以分析，所以將新的演算法以模擬的方式來驗證，是目前較常用的方法。另一種情況是，當要架設一個新的網路環境時，必須事先評估網路的拓撲及頻寬是否足夠應付內部及外部的使用者，此時就需要一套支援模組豐富的網路模擬軟體，藉由事先模擬各種不同的網路環境，作爲決策時的參考。目前有相當多

的網路模擬軟體，表一針對目前常見的四套進行比較，其中前三套是商業軟體，第四套則是本文介紹的 ns，從中慎選一套是很重要的。

Layer	**OPNET**[1]	**BONeS**[2]	**COMNET Ⅲ**[3]	**ns**[4]
Application Layer	Database, E-mail, FTP, HTTP, MTA, Remote login, Print, Voice Application, Video Conferencing, X Window			HTTP, FTP, Telnet, Constant- Bit-Rate, On/Off Source
Transport Layer	TCP, UDP, NCP	TCP, UDP	ATP, NCP, TCP, NetBIOS, UDP	UDP, TCP, Fack and Asym TCP, RTP, SRM, RLM,PLM
Routing Protocols	OSPF, BGP, IGRP, RIP, EIGRP, PIM-SM		RIP, Shortest Measured Delay, OSPF, Minimum Penalty, IGRP	Session Routing, DV Routing, Centralized, dense mode, (bi-direction) shared tree mode
Network Layer	IP, IPX	IP	IP, IPX	IP
Data Link Layer	ATM, (Fast,Gigabit) Ethernet, EtherChannel, FDDI, FR, LANE, LAPB, STB, SNA, TR, X.25,802.11	ATM, Ethernet, TR, FDDI	CSMA/CD, ALOHA, TR, Token Bus, FDDI, X.25	CSMA/CD, CSMA/CA, Multihop,802.11, TDMA
Physical Layer	ISDN, SONET, xDSL		ISDN, SONET	

表一. 目前常見的網路模擬軟體中內建之模組比較

由表一可以看到 OPNET 的支援度相當廣泛，幾乎包含所有現行的網路標準，但卻需要百萬元以上，相較之下，ns 在 application layer 的支援度較少，僅支援三種真實的協定(HTTP, FTP, Telnet)，不過 ns 在 TCP 提供了相當豐富的函式庫，幾乎所有 TCP 的實作版本都有提供，再加上開放原始碼，可以任意增加修改自已想要的功能，所以許多研究機構皆已使用 ns 來進行網路模擬，並把最新的研究結果提供給 ns，所以 ns 有許多最新發展中的協定，可以與現有的協定比較或驗證其效能。

除此之外，ns 還有下列的特色：

- *Emulation*:大部份的網路模擬軟體限制所有的協定在一個模擬器內執行，ns 則提供與真實網路互動的功能。
- *Scenario generation*: 依據不同流量型態、網路架構、錯誤狀況，產生不同的測試環境。
- *Visualization*: 若只提供一些效能上的數據，並不能完全解釋結果。nam(Network

Animation)提供動畫顯示整個運作的流程，協助研究人員方便除錯。

- *Extensibility*: 同時使用兩種語言—C++, OTcl。C++用來實作核心的部分，包括事件的處理、封包的傳送等等，所以擁有較佳的效能。OTcl 用來定義、配置、控制整個模擬的過程，所以擁有較佳的彈性及互動性。

表二追本溯源 ns 的歷史，其前身是 REAL(Realistic and Large) [5]，而 REAL 是由 NEST (Network Simulation Testbed) [6]發展而來。目前 ns 每過幾天就會有更新，在 http://www.isi.edu/nsnam/ns/CHANGES.html 可以看到整個修改的 log，可以了解最新的功能。目前 ns 及內附的 nam 是 VINT(Virtual InterNetwork Testbed)計畫[7]的一部份，該計畫是由 DARPA 所贊助，合作成員包括 USC/ISI, Xerox PARC, LBNL, 以及 UC Berkeley，目的在提供完整、趨於真實的網路模擬環境。發展至今，ns 支援的平台包括大部份 Unix (FreeBSD, Linux, SunOS, Solaris)，也支援 Windows。原始碼大約有十萬行的 C++，七萬行的 OTcl ，三萬行的測試套件，二萬行的文件，已經是一個很成熟的網路模擬環境。

網路模擬軟體	NEST	REAL	ns v1.0	ns v2.0	ns v2.1b6
時間	1988	1989	Jul 31 1995	Nov 6 1996	Jan 18 2000

表二. ns 演進的歷史

2. ns 內建模組資源

在使用 ns 之前，必須了解到底 ns 提供了哪些模組可以直接使用，表三試著將內建模組加以分類列表。

Layer	Protocol	Class name in ns	Description
Application Layer	HTTP	Http/Server	Http server
		Http/Client	Http client
		Http/Cache	cache
	FTP	Application/FTP	Simulates bulk data transfer
	Telnet	Application/Telnet	Exponential or random interval
	CBR	Application/Traffic/CBR	Constant-bit-rate source
	On/Off source	Application/Traffic/<type>	<type>:Exponential, Pareto

<table>
<tr><td rowspan="22">Transport Layer</td><td>UDP</td><td>Agent/UDP</td><td>UDP sender</td></tr>
<tr><td rowspan="11">TCP (sender)</td><td>Agent/TCP</td><td>“Tahoe” TCP sender</td></tr>
<tr><td>Agent/TCP/Reno</td><td>“Reno” TCP sender</td></tr>
<tr><td>Agent/TCP/Reno/RBP</td><td>Reno TCP with Rate-based pacing</td></tr>
<tr><td>Agent/TCP/Newreno</td><td>Reno with a modification</td></tr>
<tr><td>Agent/TCP/Sack1</td><td>TCP with selective repeat (RFC 2018)</td></tr>
<tr><td>Agent/TCP/Vegas</td><td>TCP Vegas</td></tr>
<tr><td>Agent/TCP/Vegas/RBP</td><td>TCP Vegas with Rate-based pacing</td></tr>
<tr><td>Agent/TCP/Fack</td><td>Reno TCP with “forward ack”</td></tr>
<tr><td>Agent/TCP/Asym</td><td>Asymmetric bandwidth TCP sender</td></tr>
<tr><td>Agent/TCP/RFC793edu</td><td>RFC793 TCP</td></tr>
<tr><td>Agent/TCP/SackRH</td><td>TCP Rate-Halving</td></tr>
<tr><td rowspan="5">TCP(receiver)</td><td>Agent/TCPSink</td><td>TCP sink with one ACK per packet</td></tr>
<tr><td>Agent/TCPSink/DelAck</td><td>with configurable delay per packet</td></tr>
<tr><td>Agent/TCPSink/Sack1</td><td>Selective ACK sink</td></tr>
<tr><td>Agent/TCPSink/Sack1/DelAck</td><td>Sack1 with DelAck</td></tr>
<tr><td>Agent/TCPSink/Asym</td><td>Asymmetric bandwidth TCP receiver</td></tr>
<tr><td>TCP(2-way)</td><td>Agent/TCP/FullTcp</td><td>Experimental 2-way Reno TCP</td></tr>
<tr><td>RTP</td><td>Agent/RTP</td><td></td></tr>
<tr><td>RTCP</td><td>Agent/RTCP</td><td></td></tr>
<tr><td>SRM</td><td>Agent/SRM</td><td>Scalable Reliable Multicast [8]</td></tr>
<tr><td>PLM</td><td>Agent/PLM</td><td>Fast Convergence for Cumulative Layered Multicast Transmission [9]</td></tr>
<tr><td rowspan="2">Network Layer</td><td>Unicast</td><td>$ns rtproto <type></td><td><type>:static, Manual, Session, DV</td></tr>
<tr><td>Multicast</td><td>$ns mrtproto <type></td><td><type>:DM, CtrMcast, ST, BST</td></tr>
</table>

表三. ns 內建模組列表

ns 的內建模組使用方式分成兩類，一類是建立該物件，例如：

```
set tcp [new Agent/TCP]  #建立 Tahoe TCP Sender
```

另一類是以設定的方式，例如:

```
$ns mrtproto DM {}      #使用 Dense Mode Multicast routing protocol
```

ns 提供的 TCP 模組非常豐富，幾乎囊括所有的 TCP 實作版本，甚至是目前研究發展中的版本。不過，大多是單向的 sender，沒有建立及停止連線的動作 (SYN/FIN)，也無法同時雙向傳送資料，sequence number 是以封包爲單位。這些都是簡化後的 TCP，所以會有一些失真，如果模擬時需要更接近真實情況，可以考慮使用對應的 FullTcp 版本，就沒有上述的問題。

3. 模擬方法與步驟

3.1. 安裝 ns

ns 的原始程式及安裝說明位於 http://www.isi.edu/nsnam/ns/ns-build.html，目前安裝的方式分爲

兩種，一種是分別安裝各個所需套件，另一種是安裝 all-in-one 套件，建議第一次安裝 ns 可使用後者，本文以在 linux 的/home/ns 目錄下安裝 ns all-in-one 套件為例：

1. `wget http://www.isi.edu/nsnam/dist/ns-allinone-2.1b6a.tar.gz`

   #抓回最新版的 all-in-one 套件

2. `tar zxvf ns-allinone-2.1b6a.tar.gz` #解開套件

3. `cd ns-allinone-2.1b6/`

4. `./install` #若安裝出現問題，可參考 http://www.isi.edu/nsnam/ns/ns-problems.html

5. 在 PATH 環境變數加上/home/ns/ns-allinone-2.1b6/bin

6. 在 LD_LIBRARY_PATH 環境變數加上/home/ns/ns-allinone-2.1b6/otcl-1.0a5

7. `cd ns-2.1b6./validate` #檢驗是否安裝正常，時間需要半小時至一小時

因為目前最新的 all-in-one 套件是 Jan 18 2000 推出的，距今也有一段時日，這之間 ns 修正許多 bug，加入新的模組，這是使用 all-in-one 套件的缺點，所以建議可以再抓回最新的 ns daily snapshot 進行更新。可以在步驟 4.之前，先將 ns-2.1b6 的內容替換成新的 ns daily snapshot 再進行安裝。安裝完畢後，執行 ns，打 ns-version 就可以看到目前的 ns 版本：

```
ns@hades ~/ns-allinone-2.1b6 > ns
% ns-version
2.1b7-snapshot-20000803
```

ns-allinone package 是一套獨立的程式，目錄架構如圖一，所有的程式碼皆在 ns-allinone 這個目錄下，C++程式碼位於 ns-2.1 目錄，而 OTcl 的部份在 ns-2.1/tcl 目錄，test 目錄下是測試套件，ex 目錄下是一些範例程式。

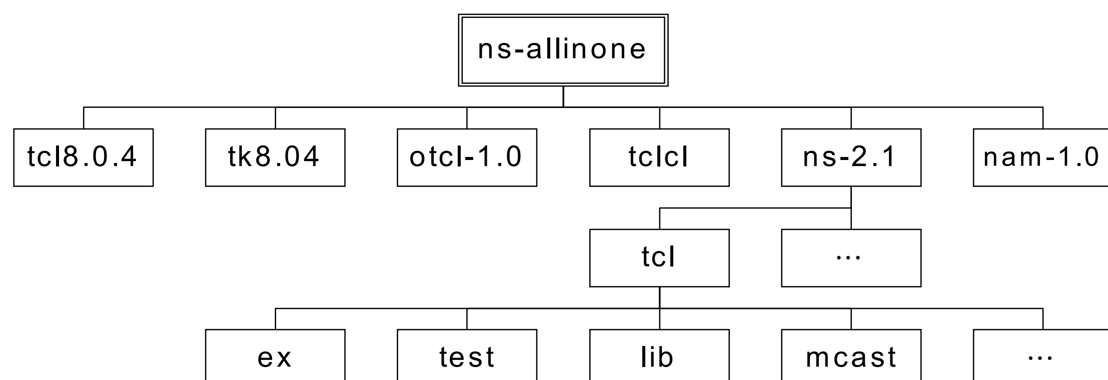


圖 一. ns 的目錄結構

3.2. ns 的內部運作

ns 的特色在於使用兩種程式語言的架構，一些比較低階的工作，例如事件的處理、封包的轉送，這些事情需要較高的處理速度，而且一旦完成就很少需要修改，所以使用 C++是最佳的選擇。另外一方面，在做研究時常需要設定不同的網路環境、動態改變協定的參數，這些事情使用像 Tcl 這類的直譯式語言將擁有較佳的彈性。ns 透過 tclcl 來連繫兩種語言之間的變數及物件，在兩種語言的特性互補下，使得 ns 成爲兼具高效能與高彈性的網路模擬軟體。

3.3. OTcl 簡介

ns 使用 MIT 發展的 OTcl (Object Tcl)做爲描述、配置、執行模擬的語言，OTcl 是 Tcl 的物件導向延伸版本。對於 C++程式設計師來說，OTcl 和 C++語法上有幾點差異性：

- C++使用”//”做爲單行註解，而 OTcl 使用”#”。
- C++僅能單一的 class 宣告，而 OTcl 中使用多次的定義。每次的定義(使用 `instproc`)增加一個 method 到該 class；每次的變數定義(使用 `set` 或在 method 中使用 `instvar`)增加一個 instance variable 到該 object。
- OTcl 中的 `init` instproc 相當於 C++中的 constructor，而 `destroy` instproc 相當於 C++中的 destructor。
- OTcl 中一定要透過 object 來呼叫，OTcl 中的 self 就相當於 C++中的 this。

在使用 ns 進行網路模擬之前，必須先學會這個語言，這節以一個小程式來介紹 OTcl 語法。

```
Class dad        #宣告一個 class dad
dad instproc greet {} {        #在 dad 增加一個 method greet
        $self instvar age_       #在 dad 增加一個 instance variable age_
        puts “$age_ years old dad: How are you doing?”     # $arge_爲變數的值
}

Class kid -superclass dad   #宣告一個 class kid 繼承 dad
kid instproc greet {} {
        $self instvar age_
          puts “$age_ years old kid: What’s up?”
}

set a [new dad]     #產生一個 dad 的 object 並設定給變數 a
$a set age_ 45     #設定 a 中的變數 age_爲 45
set b [new kid]
$b set age_ 15
$a greet              #執行 a 的 greet method
$b greet
```

程式的執行結果：

```
45 years old dad: How are you?
15 years old kid: What's up?
```

dad 執行自己的 greet() method，存取自己的 age_變數，所以印出來的結果是 45 歲，以及問候"how are you?"，同樣地 kid 也印出自己的訊息。以上簡單的介紹這兩個例子，應該可以大致了解 OTcl 的語法，若需要進一步的資料可以參考 ns/otcl/doc 目錄下有一份 OTcl Tutorial，是一份不錯的入門文件。

3.4. 模擬的步驟

以 ns 進行模擬大致可分爲三個步驟:

a.建立 network model: 描述整個網路的拓撲、頻寬等資訊。

b.建立 traffic model: 描述所有的網路流量或錯誤情況的時間、類型、或呈何種數學分佈。

c.追蹤分析結果: 模擬完成後，可藉由 nam 觀察整個流程，或是將 nam file 中想要的資訊抽取出來加以分析。

a. Network model

a) 建立網路拓撲

建立 ns 物件: `set ns [new Simulator]`

建立節點: `set n0 [$ns node]`

建立連結: `$ns duplex-link $n0 $n1 <bandwidth> <delay> <queue_type>`其中 `<queue_type>`可以是 `DropTail, RED, CBQ, FQ, SFQ, DRR`

建立 LAN: `$ns make-lan <node_list> <bandwidth> <delay> LL Queue/DropTail MAC/802.3 Channel`

若只想建立點對點的網路，則不用建立 LAN 這個步驟。

b) 選擇路由方式

- Unicast

  `$ns rtproto <type> (<type>: Static, Session, DV)`

- Multicast

  `$ns multicast (right after [new Simulator])`

  `$ns mrtproto <type> (<type>: CtrMcast, DM, ST, BST)`

b. Traffic model

a) 建立連線

- TCP

  `set tcp [new Agent/TCP]   #建立 Tahoe TCP sender`

```
set tcpsink [new Agent/TCPSink]  #建立receiver
$ns attach-agent $n0 $tcp  #把TCP sender接在n0節點上
$ns attach-agent $n1 $tcpsink  #把receiver接在n1節點上
$ns connect $tcp $tcpsink  #建立連線
```

- UDP

```
set udp [new Agent/UDP]  #建立UDP sender
set null [new Agent/NULL]  #建立receiver
$ns attach-agent $n0 $udp  #把UDP sender接在n0節點上
$ns attach-agent $n1 $null  #把reciver接在n1節點上
$ns connect $udp $null  #建立連線
```

b) 產生流量

- TCP

FTP(or Telnet)

```
set ftp [new Application/FTP]  #模擬FTP的application source
$ftp attach-agent $tcp  #將ftp source連到tcp agent
```

- UDP

CBR(or Exponential,Pareto):

```
set src [new Application/Traffic/CBR]
$src attach-agent $udp  #將source連到udp agent
```

c) 加入錯誤模組

```
set loss_module [new ErrorModel]
$loss_module set rate_ 0.01  #error rate爲1%
$loss_module unit pkt  #以封包爲單位
$loss_module ranvar [new RandomVariable/Uniform] #隨機變數uniform分配
$loss_module drop-target [new Agent/Null]  #封包丟棄點
$ns lossmodel $loss_module $n0 $n1  #在n0到n1節點之間設定error model
```

d) 建立排程

```
$ns at <time> <event> (其中<event>爲任何合法的ns/tcl命令)
$ns run  #開始執行
```

c. Tracing

```
$ns namtrace-all [open test.nam w]  #以nam格式追蹤所有封包
```

test.nam的內容包括網路的拓撲(node,link,queues,..)，以及所有封包的資訊，nam使用這個檔來展現整個模擬的過程。我們可以使用關於封包的資訊來做一些追蹤分析，封包資訊的格式如下：

```
<event-type> -t <time> -s <source> -d <dest> -p <pkt-type> -e <pkt-size>
-c <flow-id> -I <unique-id> -a <pkt-attribute> -x <ns-traceinfo>
```

<event-type>:'h'代表HOP，這個封包開始從source傳送到dest

'r'代表 Receive，這個封包已送達目的地

'd'代表 Drop，這個封包被丟棄

'+'代表 Enter queue

'-‘代表 Leave queue

<pkt-attribute>:代表 color id

<ns-traceinfo>:代表 source and destination node and port address, sequence number, flag, and the type of message

4.實例 I ：TCP 版本之比較

在 TCP/IP 網路上，當資料從頻寬較大的網路，送到頻寬較小的網路時，這時候就會造成網路壅塞的現象。同樣，在很多的流量同時到達一個路由器時，也會造成網路塞車。所以在 TCP layer 有 flow control 的機制來防止或減少壅塞[10]。在這個例子裡，會比較表四中各種 TCP 實作版本，在不同情況(1,2,3 個封包漏失)的效能，並示範如何使用 awk 從 nam-trace 檔中得到需要的資訊。

TCP 版本	特 色
Tahoe	Slow-Start, Congestion avoidance, and Fast Retransmit
Reno	Tahoe plus Fast Recovery
NewReno	Reno with performance modification
Sack1	TCP with selective repeat(follows RFC2018)
Fack	Reno TCP with forward ack

表四. TCP 實作版本比較[11]

Slow Start and Congestion Avoidance

這個演算法的基本假設是：封包在途中遭到損壞的機率很小(遠小於 1%)，所以只要發生封包漏失，就是網路發生壅塞的信號。辨別封包漏失的方式有兩種：發生 retransmit timeout(RTO)和收到重複的 ACK。在每個連線下需要兩個變數：cwnd 代表 congestion window，ssthresh 代表 slow start threshold size。

1. 連線起啓時，cwnd 爲一個 segment 大小，ssthresh 爲 65535 bytes。TCP 不會送超過 cwnd 和 advertised window 中較小的一個。
2. 當壅塞發生時(timeout 或收到重複的 ACK)，ssthresh 會變成目前 window size 的一半。此外，如果壅塞是因爲 timeout 的話，cwnd 會設成一個 segment，也就變成 slow start。
3. 每當收到一個 ACK，就增加 cwnd，可是到底增加多少要看現在是 slow start 還

是 congestion avoidance。當 cwnd 小於 ssthresh 時，是 slow start，反之則是 congestion avoidance。

4. 在 Congestion avoidance 時，每當收到一個 ACK，cwnd 增加 1/cwnd。所以 slow start 是呈指數成長，而 congestion avoidance 是呈線性成長。

Fast Retransmit and Fast Recovery Algorithms

當 TCP 收到一個順序錯誤的封包時，它必須立刻送出一個重複的 ACK，告訴傳送方它所等待的封包編號。因爲我們並不知道這個重複的 ACK 是因爲封包漏失，或只是到達順序的不同所引起，所以必須繼續等待其他的 ACK。因此，這個演算法有一個基本的假設：如果只是一個到達順序錯誤的封包，那麼只會有一個或兩個的重複 ACK，假如收到三個以上重複的 ACK，表示非常有可能發生雍塞。這個時候立刻重送這個封包，而不必等到 RTO，這就是 fast retransmit algorithm。接著跳過 slow start，而直接進行 congestion avoidance，這就是 fast recovery algorithm。詳細的說明如下：

1. 當收到第三個重複的 ACK，把 ssthresh 設爲目前 window 大小的一半，重送這個封包。
2. 設定 cwnd 爲 ssthresh + 3 * segment size
3. 每次收到另一個重複的 ACK，增加 cwnd 一個 segment size，然後送出一個封包(如果在新的 cwnd 允許範圍內)。
4. 當收到新的 ACK 時，設定 cwnd 和 ssthresh 一樣大。

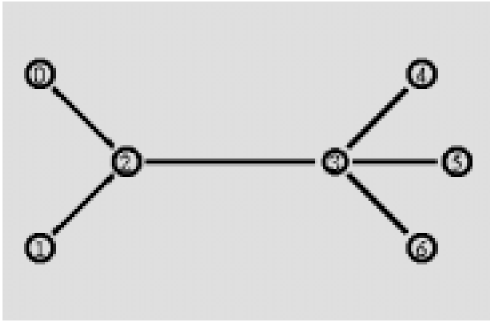
SACK [12]

當同一個 window 內漏失多個封包時，因爲使用 cumulative ack 的關係，每個 round trip time 只會知道一個封包漏失，所以原始的 TCP 可能會有極差的效能。Sack Option header 可記錄幾個區域，分別代表尚未收到的封包號碼，可以明確地通知傳送方哪幾個封包沒有收到。

FACK [13]

SACK Option 有記錄目前接收方收到封包中最高的編號，稱之爲 *forward ack*。相對於其他的 TCP 以重複的 ack 來估計網路上的資料數，FACK 演算法使用 SACK Option 來明確的計算網路上的封包數，當小於 cwnd 時就可以繼續傳送資料。

網路上的封包數＝下一個要傳送的封包編號－fack ＋ 重送的封包數



- 節點 2 至 3 的 link 爲瓶頸
- 其餘 link 皆是頻寬 10Mb，10ms delay，DropTail Queue
- FTP 連線從節點 0 至節點 5
- UDP 從節點 1 multicast 至節點 4, 6

圖二. 實例的網路拓撲

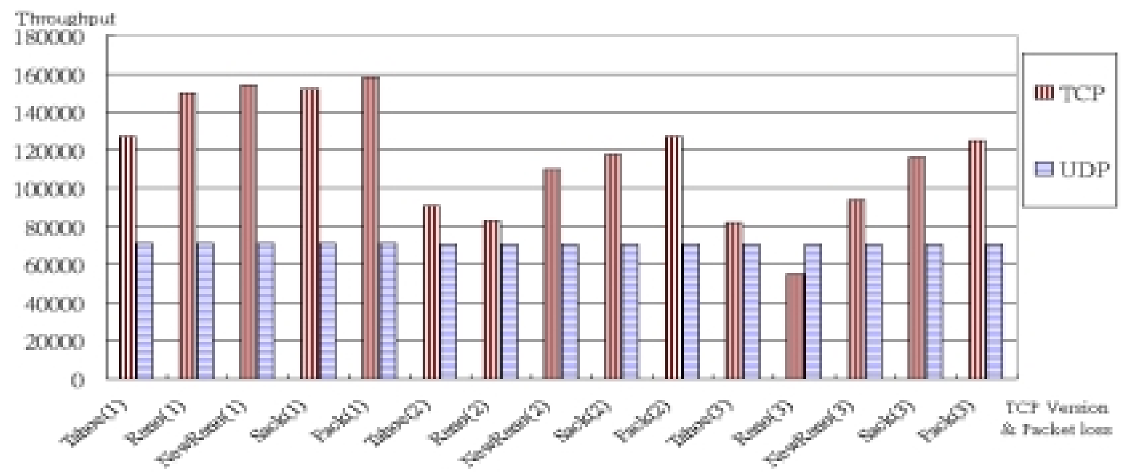
實例一的網路拓撲如圖二，而其 OTcl 程式碼如下：

```
set ns [new Simulator -multicast on]  #建立 scheduler 並允許 multicast
$ns color 1 red    #第一條 flow 在 nam 中使用紅色表示
$ns color 2 blue   #第二條 flow 在 nam 中使用藍色表示
for {set i 0} {$i < 7} {incr i} {   # i 從 0 而 6 的 for 迴圈，建立 n0 到 n6 共七個節點
   set n($i) [$ns node]
}
set nf [open out.nam w]   # 產生 nam trace 檔
$ns namtrace-all $nf
#建立六個 links
$ns duplex-link $n(0) $n(2) 10Mb 10ms DropTail
$ns duplex-link $n(1) $n(2) 10Mb 10ms DropTail
$ns duplex-link $n(2) $n(3) 8.5Mb 25ms RED     #瓶頸 link
$ns duplex-link $n(3) $n(4) 10Mb 10ms DropTail
$ns duplex-link $n(3) $n(5) 10Mb 10ms DropTail
$ns duplex-link $n(3) $n(6) 10Mb 10ms DropTail
$ns queue-limit $n(2) $n(3) 4       #設定瓶頸 link 的 queue buffer size 爲四個封包
$ns duplex-link-op $n(2) $n(3) queuePos +0.5    #設定 nam 中 queue 的位置
$ns mrtproto DM {} #使用 Dense mode multicast

set tcp [new Agent/TCP]
set sink [new Agent/TCPSink]
$ns attach-agent $n(0) $tcp
$ns attach-agent $n(5) $sink
$ns connect $tcp $sink
$tcp set class_ 1    #設定 flow id 爲 1
$tcp set window_ 20   #設定 window 大小爲 20 個封包
set ftp [new Application/FTP]
$ftp attach-agent $tcp
#建立 UDP 連線從節點二至 multicast group
set group [Node allocaddr]
set udp [new Agent/UDP]
$ns attach-agent $n(1) $udp
$udp set dst_addr_ $group
$udp set dst_port_ 0
$udp set class_ 2
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
set rcvr0 [new Agent/Null]
$ns attach-agent $n(4) $rcvr0
set rcvr1 [new Agent/Null]
$ns attach-agent $n(6) $rcvr1
#建立 schedule
$ns at 0.0 "$n(4) join-group $rcvr0 $group"    #將節點 4,6 加入 multicast group
$ns at 0.0 "$n(6) join-group $rcvr1 $group"
$ns at 0.2 "$cbr start"
$ns at 0.2 "$ftp start"
$ns at 1.5 "finish"
proc finish {} {
     global ns nf
     $ns flush-trace
     close $nf
     exec nam out.nam &
     exec awk {
         {
```

```
            if($1=="r" && $15==3 && $7==5 && $9=="tcp")
            print $3, $2
          }
      } out.nam > tahoe  #將節取到的資料放到檔案 tahoe
      exit 0
}
$ns run   #開始執行
```

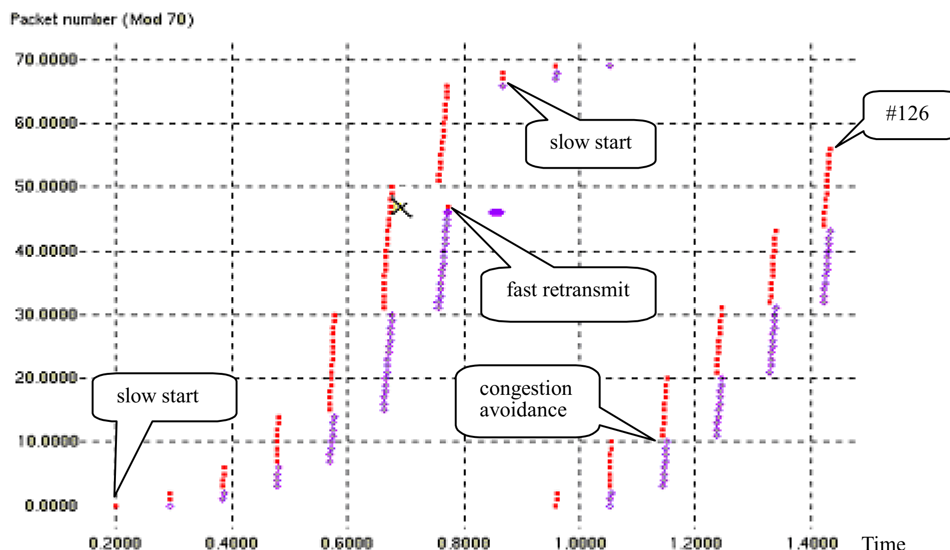
重覆執行不同版本的 TCP，以及不同的網路環境，畫出圖三的長條圖。值得注意的是，Reno TCP 在有兩個 packet loss 以上的情況，效能會比 Tahoe TCP 來得更差，就讓我們來一探究竟，看看不同封包漏失數下各個 TCP 版本的表現。



圖三. 各種 TCP 版本在漏失封包時的效能比較(括號的數字代表幾個封包漏失)

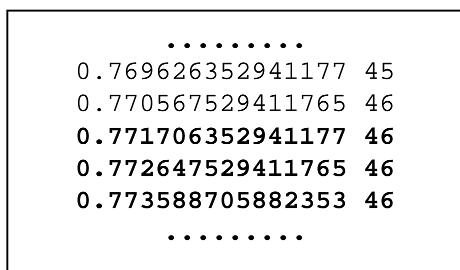
一個封包漏失

首先，來觀察一下 Tahoe TCP 在一個封包漏失(在 0.68 秒)的反應。在圖四中的 X 軸代表時間，Y 軸代表的是封包編號(Modulus 70)，實心的方格代表封包傳送，x 代表封包漏失，空心的菱形代表 ack。在圖四可以清楚地看到在 0.2 秒時送出第一個封包，在 0.29 秒收到一個 ack 後再送出兩個封包，在 0.38 秒收到兩個 ack 後再連續送出四個封包，繼續呈指數成長 8 個、16 個，這段時間就是所謂的 ***slow-start algorithm***。0.68 秒封包漏失後，發送端察覺到有壅塞發生，立刻將 slow start threshold 降爲 congestion window 的一半，congestion window 降到 1，再開始重新執行 slow start。

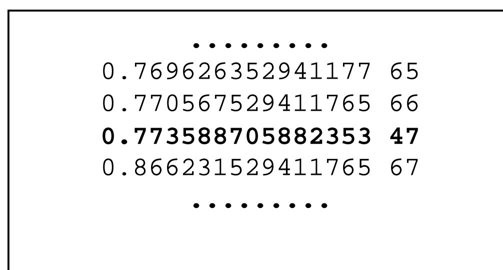


圖四. 在一個封包漏失時，Tahoe TCP 的封包傳送時間

圖五和圖六每行的第一個數字是時間，第二個數字是編號。由這兩個圖可以看到，當發送端在 0.7735 秒收到第三個重複的 ack，就知道發生了封包漏失，不等 retransmit timer timeout 立刻重送第 47 號的封包，這就是 fast-retransmit algorithm。


....
0.769626352941177 45
0.770567529411765 46
0.771706352941177 46
0.772647529411765 46
0.773588705882353 46
....

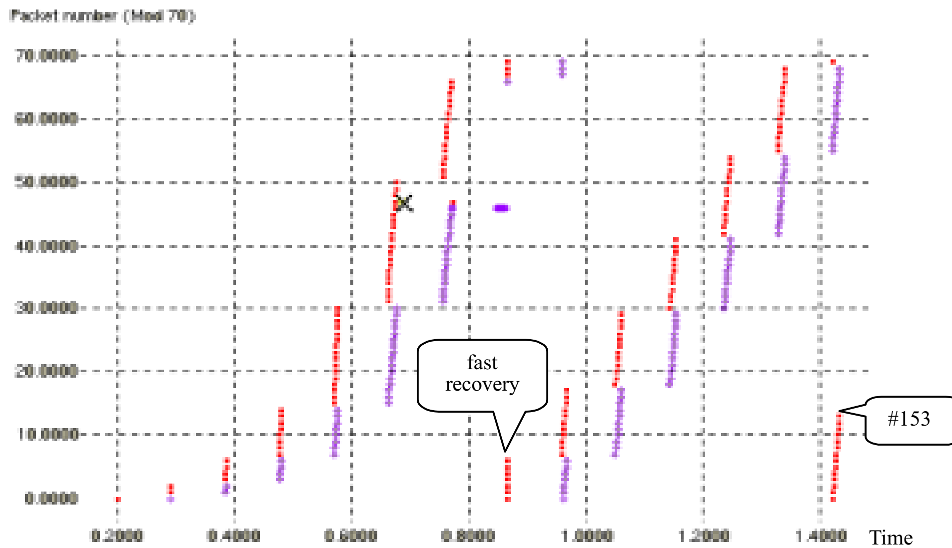
圖五. Tahoe TCP 封包發送時間


....
0.769626352941177 65
0.770567529411765 66
0.773588705882353 47
0.866231529411765 67
....

圖六. Tahoe TCP ack 接收時間

Reno TCP 比 Tahoe 增加了 ***fast recovery algorithm***，其做法是在封包漏失發生後，congestion window 設為原來的一半，然後直接執行 congestion avoidance，目的在於不要讓 congestion window 一下子縮得太小了，而影響了效能，在此例可以發現在同樣的時間裡，圖七 Reno TCP 傳送至第 153 個封包，而圖六的 Tahoe TCP 僅傳送至第 126 個封包。

在一個封包漏失的情況下，New-Reno,Sack1,Fack 的表現和 Reno 差不多。

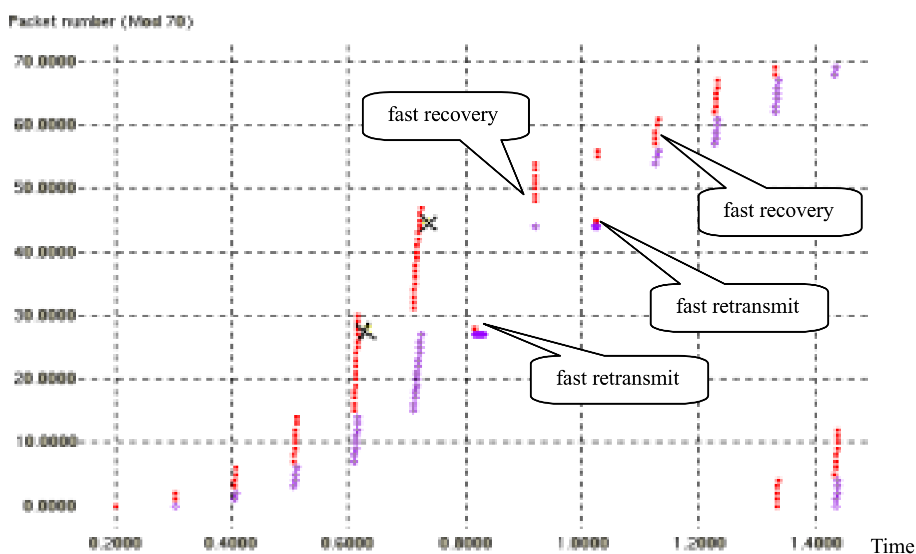


圖七. 在一個封包漏失時，Reno TCP 的封包傳送時間

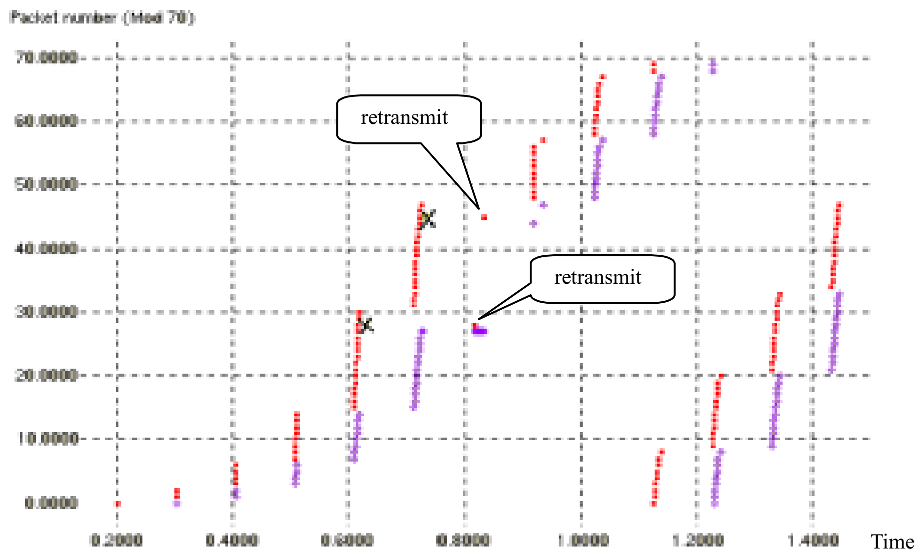
二個封包漏失

在遇到兩個封包漏失時，Tahoe TCP 一樣經由 slow start 慢慢恢復傳送的速度。在圖八可以發現，Reno TCP 經過連續兩次的 fast retransmit 及 fast recovery 來復原，不過每次把 cwnd 減半兩次，所以傳送速度慢了許多。

由圖九可以看到 Sack TCP 在 0.81 秒收到三個重複的封包後，透過 Sack Option 知道第 28, 45 號封包皆未送達。所以 fast-retransmit 第 28 個之後，立刻送出漏失的 45 號封包，但必須等到 0.91 秒有新的 ack 到達才能開始繼續傳送下一個封包。

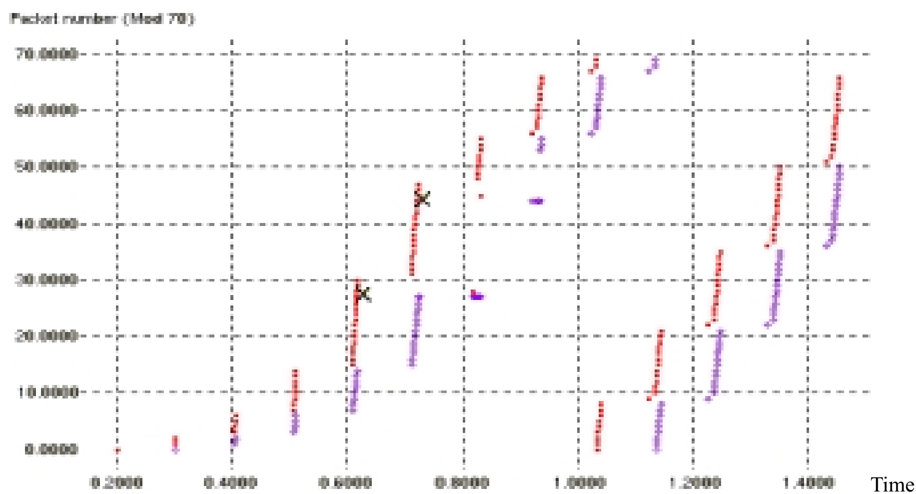


圖八.兩個封包漏失時，Reno TCP 的封包傳送時間



圖九. 兩個封包漏失時，Sack TCP 的封包傳送時間

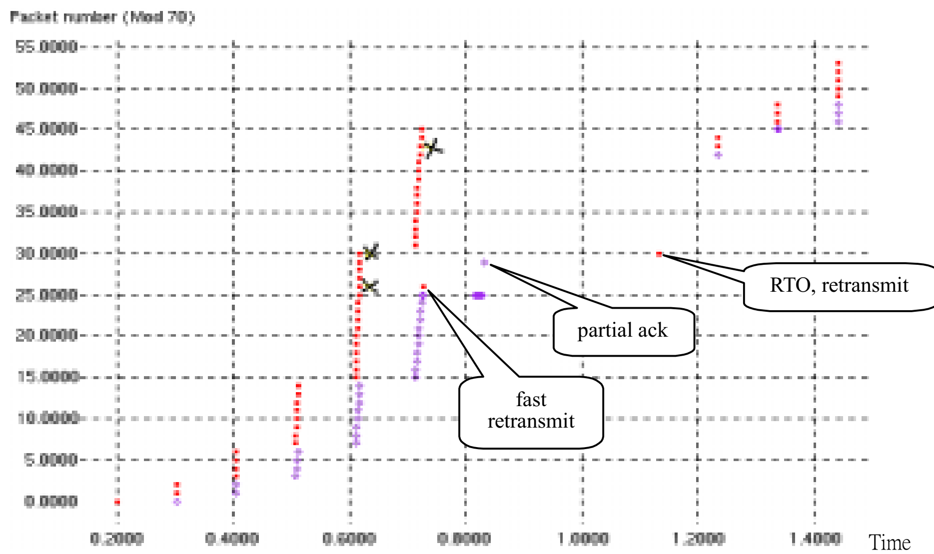
在圖十可以發現 Fack TCP 在 0.81 秒的時候收到三個重複的封包後，fast retransmit 第 28 號編包，接著在 0.83 和 Sack TCP 一樣立刻補送第 45 號封包，不過 Fack TCP 估計網路上的未送達的封包仍小於 cwnd，而繼續傳送新的封包，所以效能上更佳。



圖十. 兩個封包漏失時，Fack TCP 的封包傳送時間

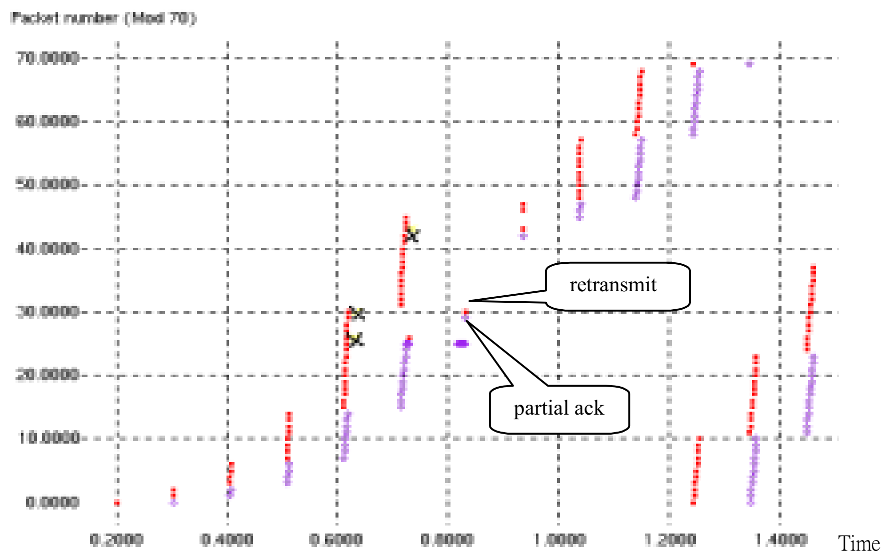
三個封包漏失:

在同一個 window 內有兩個以上的封包漏失時，如圖十一中的第 26, 30 號封包漏失，fast-retransmit 第 26 號封包後，回覆的 ack 是 29，並沒有 ack 所以已送達的封包，這稱爲 partial ack。這個時候 Reno TCP 就必等到 1.13 秒時 RTO，才能重送第 30 號封包，所以會有嚴重的效能低落。



圖十一. 三個封包漏失時，Reno TCP 的封包傳送時間

上述的 Reno TCP 的問題，在 NewReno 中就改善了。如圖十二所示，在遇到 partial ack 29 的時候，NewReno TCP 會預期 30 號封包已經漏失，所以立即重送，而不必等到 RTO。



圖十二. 三個封包漏失時，NewReno TCP 的封包傳送時間

Sack 及 Fack 在漏失三個封包的情況下，一樣表現得比其他的 TCP 版本來得優異。

5.實例 II ：新增協定比較 TCP 啓始視窗大小

這個實例示範如何增加一個新的 TCP agent (IW TCP)，這個 Agent 的 initial window size 並不是預設的 1，而是可以更改爲想要的大小。主要目的是觀察 initial TCP window size(IW)對 TCP performance 的影響。IW 一直是 TCP Implementation Working Group(tcpimpl)[14]的一個重要討論

議題，RFC 2414、RFC 2581 有相關的討論，目前的 TCP 實作將 IW 設為一個 segment。在這個實例中，將分別將 IW 設為 2、4、8 個封包，觀察其對 TCP 效能的影響。

首先，必須曉得 ns 透過表五列出的 class 來達成 C++與 OTcl 兩種語言並存，並且可以互相溝通，若是要在 ns 裡新增功能時必須注意：

- 決定適當的繼承點，例如要增加新的 TCP，則可以繼承自 Agent/TCP
- 建立新的 class
- 定義與 OTcl 的連結
- 撰寫相關的 OTcl code 去存取

TclObject	大部分 ns 物件的 base class
	bind(): 連結 C++與 OTcl 中的變數
	command(): 將 OTcl 中的 method 連結至 C++
TclClass	產生 TclObject 物件
Tcl	提供包裝好的 OTcl 直譯器，使得 C++ method 可以存取
TclCommand	提供可以直接在直譯器中執行的命令，例如 ns-version
EmbeddedTcl	讀取及執行初始化的 script 命令

表五. C++ and OTcl Linkage

實例二示範如何增加一個新的 Tcp agent，這個 agnet 可以方便地設定 IW 值。

```
#include "tcp.h"
#include "tclcl.h"

class IWTcpAgent : public TcpAgent { //繼承自 TcpAgent
public:
       IWTcpAgent();
       virtual void set_initial_window() { //設定 initial window size
            cwnd_ = ini_win_;
       }
private:
       int ini_win_;  //initial window size
};
```

圖十三. tcp-iw.h

```
#include "tcp-iw.h"

static class IWTcpClass : public TclClass {
public:
     IWTcpClass() : TclClass("Agent/TCP/IW") {} //定義相對應的 OTcl name
     TclObject* create(int, const char* const*) { //建立相對應的 OTcl 物件
          return (new IWTcpAgent());
     }
```

```
} tcpjs_agent;

IWTcpAgent::IWTcpAgent() {
    bind("Ini_Win", &ini_win_);  //連結OTcl中的Ini_win與C++中的ini_win_
}
```

圖十四. tcp-iw.cc

將上述兩個檔案放在 ns-2 的目錄下，修改 Makefile:

```
OBJ_CC = \
        ….
        tcp-iw.cc
```

重新編譯

```
make depend
make
```

定義 ini_win 的預設值爲 1，修改 ns/tcl/lib/ns-default.tcl，增加一行

```
Agent/TCP/IW set Ini_Win 1
```

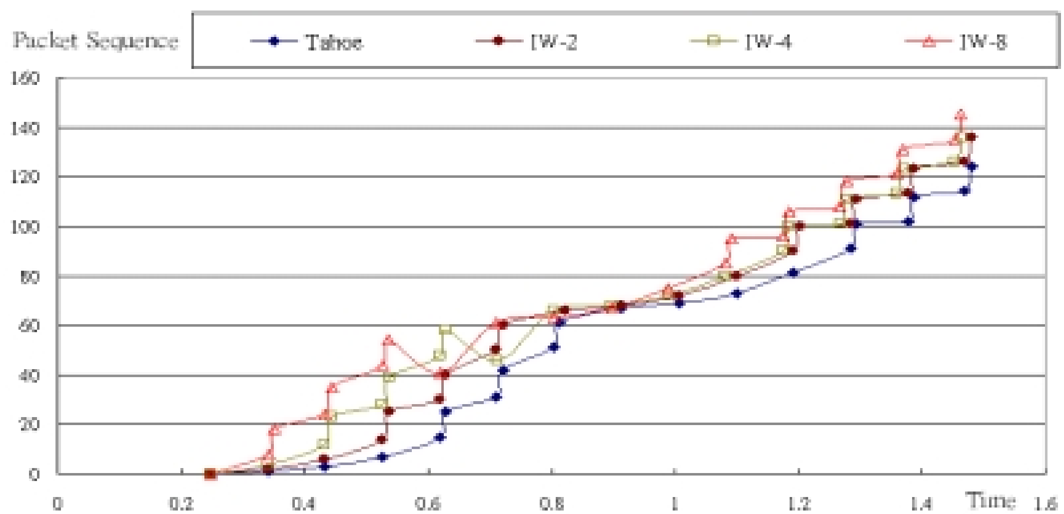
這樣就在 ns 中新增一個 agent，重做一次實例 I，這次比較一般的 Tahoe Tcp (IW=1)和 IW 等於 2、4、8，在不同數量封包漏失時的效能。

一個封包漏失: IW-8 > IW-4 = IW-2 > Tahoe

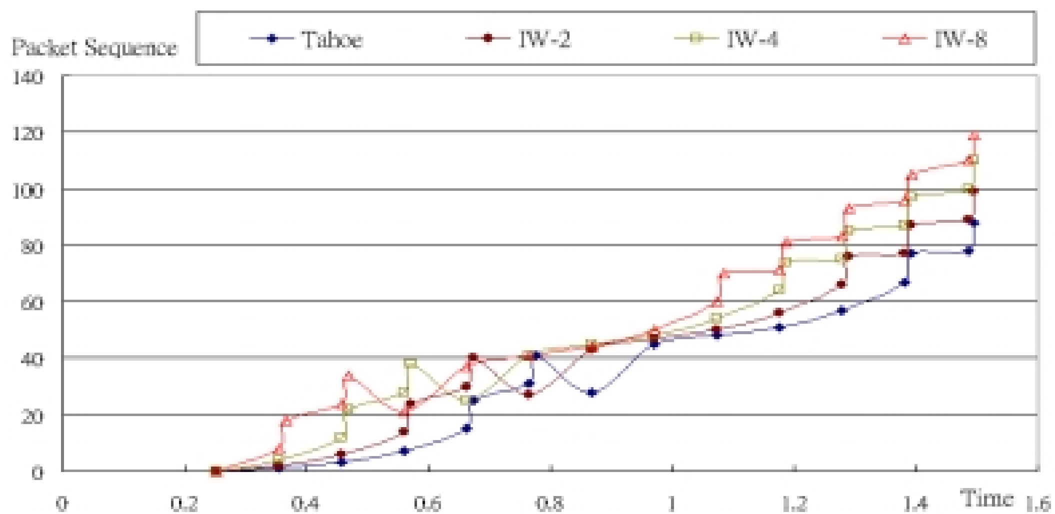
二個封包漏失: IW-8 > IW-4 > IW-2 > Tahoe

三個封包漏失: IW-4 > IW-8 > IW-2 > Tahoe

由此可以證明在封包漏失量少時，增加 IW 的值的確可以增加 TCP 的效能，不過在不同的擁塞情況下 IW 的值應該設爲多少，則有待進一步的實驗。



圖十五.漏失一個封包



圖十六. 漏失兩個封包的情況.

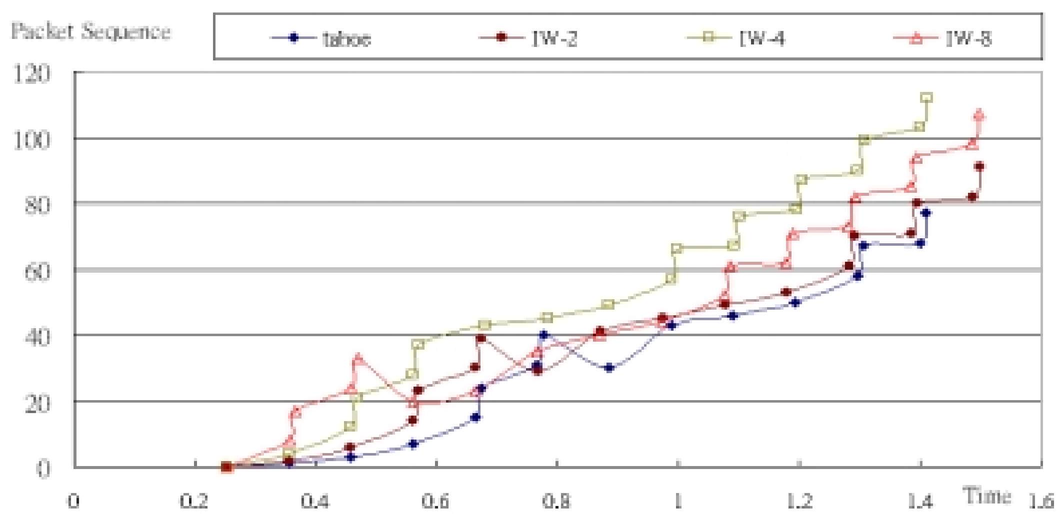


圖 十七. 漏失三個封包的情況

六、結論

ns 是一套開放原始碼且免費的網路模擬軟體，內部使用 C++做爲核心，有快速的處理速度；使用 OTcl 做爲模擬設定的語言，有著高度的使用彈性，可以輕易地使用各項內建模組。 在實例 I 中可以清楚地了解各項 TCP 流量控制的演算法，並了解 Sack 及 Fack 對 TCP 效能有極大的幫助。在實例 II 中可以發現，在封包漏失量不大時，加大 IW 對 TCP 也有正面的效果，但到底 IW 值該定爲多少，還有待進一步深入的研究。愈來愈多的研究機構使用 ns 做爲網路模擬器，有著較高的公信力，應當是研究網路相關協定時不可或缺的工具。

參考資料:

[1] OPNET Technologies, Inc. http://www.opnet.com/

[2] BONeS, http://www.cadence.com/

[3] CommnetⅢ, http://www.caciasl.com/

[4] UCB/LBNL/VINT Network Simulator – ns, http://www.isi.edu/nsnam/ns/

[5] REAL 5.0, http://www.cs.cornell.edu/skeshav/real/overview.html

[6] NeST2.6, ftp://ftp.cs.columbia.edu/nest/

[7] Virtual InterNetwork Testbed project, http://www.isi.edu/nsnam/vint/index.html

[8] http://www.aciri.org/floyd/srm.html

[9] http://www.eurecom.fr/~legout/Research/research.html

[10] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC 2001, Jan 1997

[11] Kevin Fall and Sally Floyd. "Simulation-based Comparisons of Tahoe, Reno, and Sack TCP", Computer Communication Review, Jul 1996

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, Oct, 1996

[13]M. Mathis, J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control," Proceedings of SIGCOMM'96, pp. 281-191, August, 1996, Stanford, CA.

[14] TCP Implementation Working Group, “TCP Implementation (tcpimpl) Charter”, http://www.ietf.org/html.charters/tcpimpl-charter.html, Jul. 2000