

實驗

# 3

## TCP 與 UDP 模擬實驗

### 實驗目的

- ① 瞭解如何在 NS2 建立 TCP 連線與 UDP 連線。
- ② 如何把模擬過程輸出到檔案，最後藉由工具的分析把結果顯示出來。

01000100101010  
0011100101010  
11001010100101000101010  
10100010010101  
010001001010  
01010001010101000100101010  
0100101010001010101000100101010

## 背景知識

NS2 網路模擬程式(TCL script)的架構大致上都如下面程式所示，所以在未來的實驗都會有類似底下的程式碼。

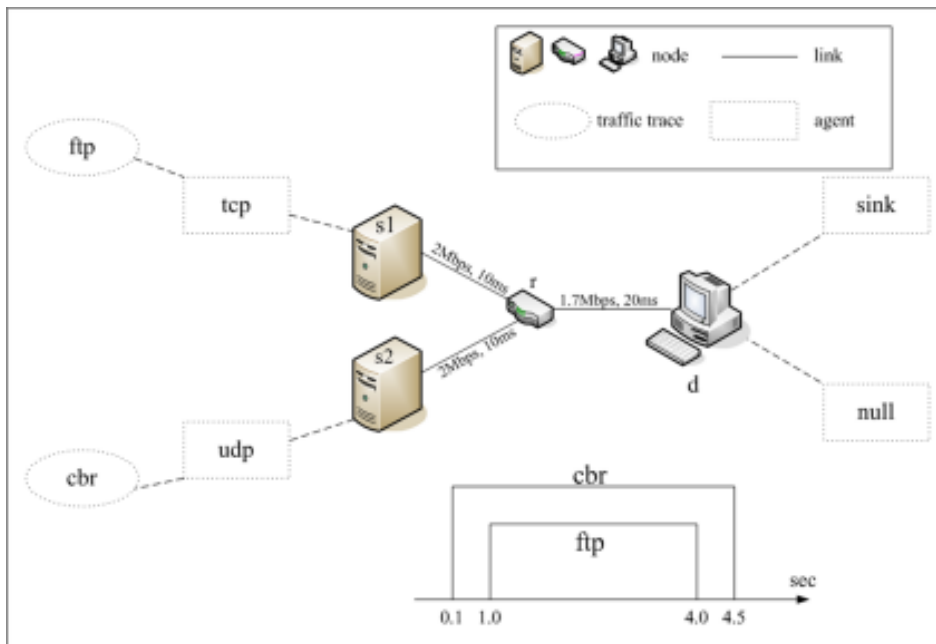
```
# "#" 後面的文字是備註說明, 所以 NS2 不會執行此行
# 產生一個模擬的物件
set ns [new Simulator]
# 定義一個結束的程序
proc finish {} {
    exit 0
}
# 底下可以新增一些如網路架構和應用程式設定的程式碼
# 在適當的時間去呼叫 finish 程序, 以結束模擬, 底下的例子是在第 5 秒的時候
$ns at 5.0 "finish"
# 開始執行模擬
$ns run
```

## 實驗步驟

在這個實驗中，將以一個簡單的網路環境為範例，練習如何在模擬結束後，使用一些工具(awk 和 gnuplot)來分析和呈現模擬結果，呈現的效能評比指標包括了端點到端點的延遲(End-to-End Delay)、抖動率(Jitter)、封包遺失率(Packet Loss)和吞吐量(Throughput)。分析所採用的方法則是去分析網路模擬過程記錄檔(traffic trace file)的方式。

## 一、網路模擬範例

■ 模擬的網路架構圖。



這個網路的環境包含了兩個傳輸節點 s1 和 s2，路由器 r，和資料接收端 d。s1 到 r 之間與 s2 到 r 之間的網路頻寬都是 2Mbps，傳遞延遲時間是 10ms。網路架構中的頻寬瓶頸是在 r 到 d 之間，頻寬為 1.7Mbps，傳遞延遲的時間為 20ms。所有鏈路的佇列管理機制都是採用 DropTail，且 r 到 d 之間的最大佇列長度(queue limit)是 10 個封包的長度。在 s1 到 d 之間會有一條 FTP 的連線，FTP 應用程式是架構在 TCP 之上，所以在寫模擬環境的描述語言的時候，必需先建立一條 TCP 的連線，因此在來源端 s1 上使用 TCP agent 產生” tcp” 來發送 TCP 的封包；在目的地 d 使用 TCP sink agent 產生” sink” 來接受 TCP 的資料，並產生回覆封包(ACK)回傳送端，最後把接收的 TCP 封包釋放。另外，還須要把這兩個 agent 連起來(connect)，連線才能建立。

若是沒有額外的參數設定，TCP 封包的長度為 1 Kbytes。此外，若想查看 NS2 模擬參數內定值設定，可以在 `~\ns-allinone-2.27\ns-2.27\tcl\lib` 目錄下的 `ns-default.tcl` 找到。另外，在 s2 到 d 之間有一條固定的傳輸速率的連線(Constant Bit Rate, CBR)，CBR 應用程式是架構在 UDP 之上，因此必需在 s2 使用 UDP agent 來產生”udp”用來發送 UDP 封包，在 d 上使用 Null agent 來產生”sink”以接收由 s2 傳送過來的 UDP 封包，然後再把接收的封包釋放。CBR 的傳送速度為 1Mbps，每一個封包大小為 1Kbytes。CBR 是在 0.1 秒開始傳送，在 4.5 秒結束傳輸；FTP 是在 1.0 秒開始傳送，4.0 秒結束傳輸。

■ TCL 程式碼。(光碟中的 Exp3\exp3.tcl)

```
# 產生一個模擬的物件
set ns [new Simulator]
# 針對不同的資料流定義不同的顏色，這是要給 NAM 用的
$ns color 1 Blue
$ns color 2 Red
# 開啟一個 NAM 記錄檔
set nf [open out.nam w]
$ns namtrace-all $nf
# 開啟一個模擬過程記錄檔，用來記錄封包傳送的過程
set nd [open out.tr w]
$ns trace-all $nd
# 定義一個結束的程序
proc finish {} {
    global ns nf nd
    $ns flush-trace
    close $nf
    close $nd
    # 以背景執行的方式去執行 NAM
    exec nam out.nam &
    exit 0
}
```

```
}  
# 產生傳輸節點, s1 的 id 為 0, s2 的 id 為 1  
set s1 [$ns node]  
set s2 [$ns node]  
# 產生路由器節點, r 的 id 為 2  
set r [$ns node]  
# 產生資料接收節點, d 的 id 為 3  
set d [$ns node]  
#s1-r 的鏈路具有 2Mbps 的頻寬, 10ms 的傳遞延遲時間, DropTail 的佇列管理方式  
#s2-r 的鏈路具有 2Mbps 的頻寬, 10ms 的傳遞延遲時間, DropTail 的佇列管理方式  
#r-d 的鏈路具有 1.7Mbps 的頻寬, 20ms 的傳遞延遲時間, DropTail 的佇列管理方式  
$ns duplex-link $s1 $r 2Mb 10ms DropTail  
$ns duplex-link $s2 $r 2Mb 10ms DropTail  
$ns duplex-link $r $d 1.7Mb 20ms DropTail  
# 設定 r 到 d 之間的 Queue Limit 為 10 個封包大小  
$ns queue-limit $r $d 10  
# 設定節點的位置, 這是要給 NAM 用的  
$ns duplex-link-op $s1 $r orient right-down  
$ns duplex-link-op $s2 $r orient right-up  
$ns duplex-link-op $r $d orient right  
# 觀測 r 到 d 之間 queue 的變化, 這是要給 NAM 用的  
$ns duplex-link-op $r $d queuePos 0.5  
# 建立一條 TCP 的連線  
set tcp [new Agent/TCP]  
$ns attach-agent $s1 $tcp  
set sink [new Agent/TCPSink]  
$ns attach-agent $d $sink  
$ns connect $tcp $sink  
# 在 NAM 中, TCP 的連線會以藍色表示  
$tcp set fid_ 1
```

```
# 在 TCP 連線之上建立 FTP 應用程式
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP
# 建立一條 UDP 的連線
set udp [new Agent/UDP]
$ns attach-agent $s2 $udp
set null [new Agent/Null]
$ns attach-agent $d $null
$ns connect $udp $null
# 在 NAM 中，UDP 的連線會以紅色表示
$udp set fid_ 2
# 在 UDP 連線之上建立 CBR 應用程式
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
# 設定傳送封包的大小為 1000 byte
$cbr set packet_size_ 1000
# 設定傳送的速率為 1Mbps
$cbr set rate_ 1mb
$cbr set random_ false
# 設定 FTP 和 CBR 資料傳送開始和結束時間
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"
# 結束 TCP 的連線 (不一定需要寫下面的程式碼來實際結束連線)
$ns at 4.5 "$ns detach-agent $s1 $tcp"
$ns at 4.5 "$ns detach-agent $d $sink"
# 在模擬環境中，5 秒後去呼叫 finish 來結束模擬 (這樣要注意模擬環境中
```

```
# 的 5 秒並不一定等於實際模擬的時間)
$ns at 5.0 "finish"
# 執行模擬
$ns run
```


■ 執行的方法。

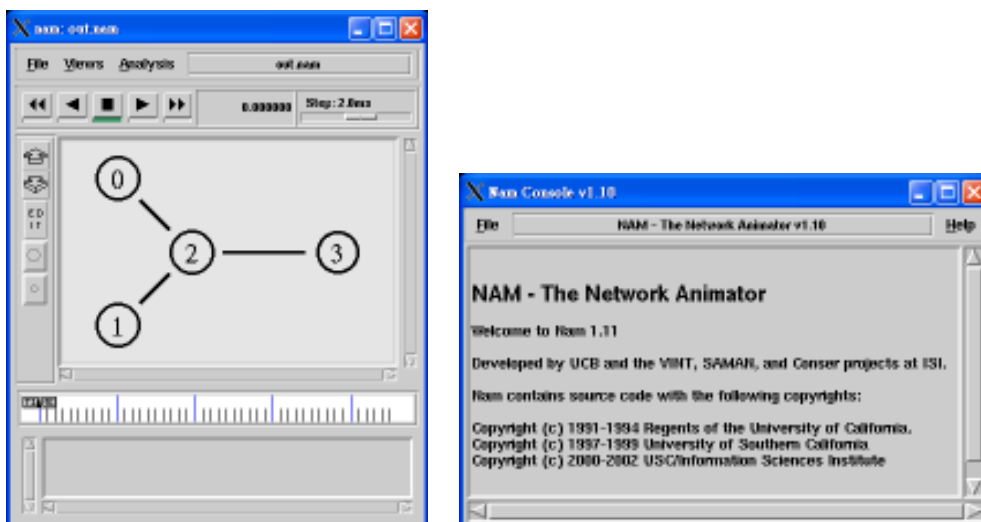
(\$ 是在 cygwin 下的提示符號)。在執行此 TCL script，請先進入繪圖模式。

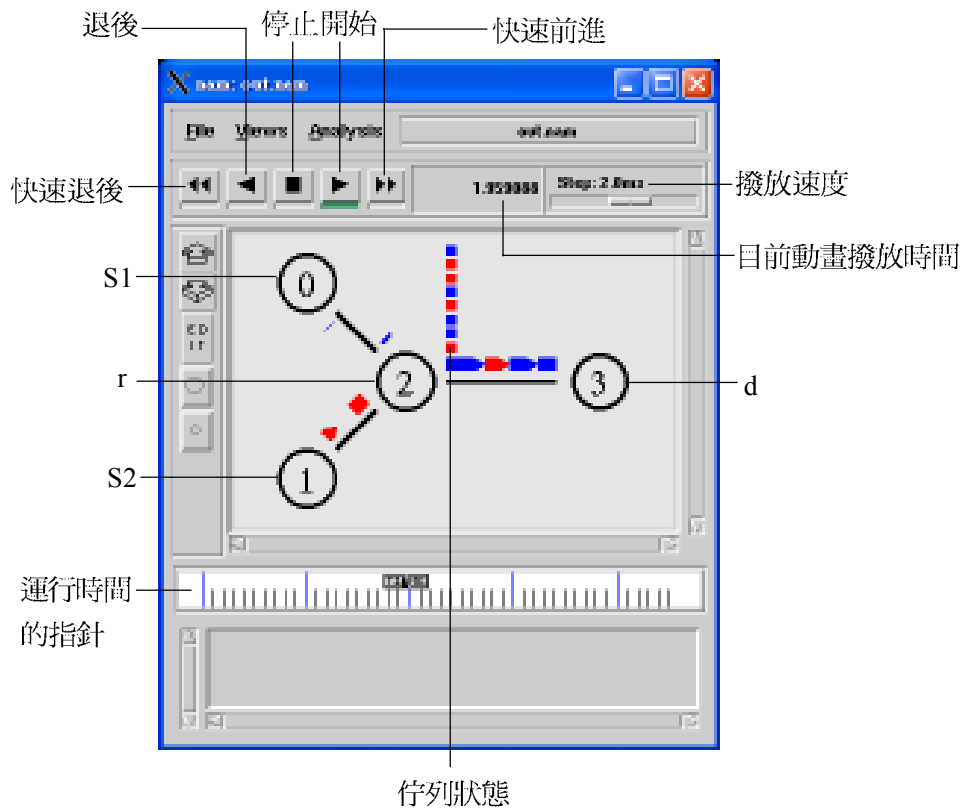
```
$ startxwin.bat
$ ns exp3.tcl
```

■ 執行的結果。

在與 exp3.tcl 同一個目錄下會產生一個 out.tr(模擬過程記錄檔)和一個 out.nam (NAM 記錄檔)，另外會開啟兩個新視窗，如下圖所示。

■ 執行 NAM，觀察網路模擬的過程。(請按左上圖中的 )





## 二、分析模擬結果

### 1. 模擬過程記錄檔內容與格式

模擬結束後，除了使用NAM觀看模擬的過程，另外就是要使用模擬過程記錄檔去做一些數值的分析，因此這個記錄檔很重要，所以需要好好的先瞭解這些記錄檔中記錄的格式。

以下是 out.tr 的部份記錄。

```
+ 0.1      1 2   cbr 1000  -----  2  1.0  3.1  0  0
- 0.1      1 2   cbr 1000  -----  2  1.0  3.1  0  0
+ 0.108    1 2   cbr 1000  -----  2  1.0  3.1  1  1
- 0.108    1 2   cbr 1000  -----  2  1.0  3.1  1  1
r 0.114    1 2   cbr 1000  -----  2  1.0  3.1  0  0
+ 0.114    2 3   cbr 1000  -----  2  1.0  3.1  0  0
- 0.114    2 3   cbr 1000  -----  2  1.0  3.1  0  0
+ 0.116    1 2   cbr 1000  -----  2  1.0  3.1  2  2
- 0.116    1 2   cbr 1000  -----  2  1.0  3.1  2  2
r 0.122    1 2   cbr 1000  -----  2  1.0  3.1  1  1
+ 0.122    2 3   cbr 1000  -----  2  1.0  3.1  1  1
.....
```

這個記錄的格式如下：

event	time	from	to	pkt	pkt	flags	fid	src	dst	seq	pkt id
			node	node	type	size		addr	addr	num	

r : receive (at to\_node)

+ : enqueue (at queue)

- : dequeue (at queue)

d : drop (at queue)

src\_addr : node.port

dst\_addr : node.port

每一筆記錄的開始都是封包事件發生的原因，若是 "r" 則表示封包被某個節點所接收；若是 "+" 則表示進入了佇列；若是 "-" 則表示離開佇列；若是 "d" 則表示封包被佇列所丟棄。接著的第二個欄位表示的是事件發生的時間；欄位三和欄位四表示事件發生的地點(從 from node 到 to node)；欄位五表示封包的型態；欄位六是封包的大小，欄位七是封包的旗標標註；欄位八表示封包是屬於哪一個資料流；欄位九和欄位十是表示封包的來源端和目的端，這兩個欄位的格式是 a.b，a 代表節點編號，b 表示埠號(port number)；欄位十一表示封包的序號；欄位十二表示封包的 id。

以前面模擬過程記錄檔的第一筆為例，意思就是說有一個封包 packet id 為 0，資料流 id 為 2，序號為 0，長度為 1000 bytes，型態為 CBR，它是從來源端 1.0 要到目的地 3.1，在時間 0.1 秒的時候，從節點 1(s2) 進入了節點 2(r) 的佇列中。

## 2. awk 語言

### 2.1 簡介

awk 是一種程式語言，具有一般程式語言常見的功能。awk 語言具有某些特點，如：使用直譯器(Interpreter)不需先行編譯；變數無型別之分(Typeless)，可使用文字當陣列的註標(Associative Array)等，因此，使用 awk 撰寫程式比起使用其它語言更簡潔便利且節省時間。另外，awk 還具有一些內建功能，使得 awk 擅於處理具資料列(Record)、欄位(Field)型態的資料；最後，awk 內建有 pipe 的功能，可將處理中的資料傳送給外部的 Shell 命令加以處理，再將 Shell 命令處理後的資料傳回 awk 程式，這個特點也使得 awk 程式很容易使用系統資源。

### 2.2 awk 是如何運作的

為了便於解釋 awk 程式架構以及相關的術語，以上面網路模擬過程記錄檔為例加以介紹。

### 2.2.1 名詞定義

a. 資料列：awk 從資料檔上讀取的基本單位，以記錄檔為例，awk 讀入的

第一筆資料列為 ” + 0.1 1 2 cbr 1000 ----- 2 1.0 3.1 0 0”

第二筆資料列為 “- 0.1 1 2 cbr 1000 ----- 2 1.0 3.1 0 0”

一般而言，一筆資料列相當於資料檔上的一行資料。

b. 欄位(Field)：為資料列上被分隔開的子字串。

以資料列” + 0.1 1 2 cbr 1000 ----- 2 1.0 3.1 0 0” 為例，一般而言是以空白字元來分隔相鄰的欄位。

一	二	三	四	五	六	七	八	九	十	十一	十二
+	0.1	1	2	cbr	1000	-----	2	1.0	3.1	0	0

當 awk 讀入資料列後，會把每個欄位的值存入欄位變數。

欄位變數	意 義
\$0	為一字串, 其內容為目前 awk 所讀入的資料列
\$1	代表 \$0 上第一個欄位的資料
\$2	代表 \$0 上第二欄個位的資料
.....	.....

### 2.2.2 程式主要結構

```

Pattern1      { Actions1 }
Pattern2      { Actions2 }
.....
Pattern3      { Actions3 }
    
```

一般常用”關係判斷式”來當成 Pattern。例如：

```
x > 3 用來判斷變數 x 是否大於 3  
x == 5 用來判斷變數 x 是否等於 5
```

awk 提供 c 語言常見的關係運算元，如： $>$ 、 $<$ 、 $>=$ 、 $<=$ 、 $==$ 、 $!=$  等等。

Actions 是由許多 awk 指令所構成，而 awk 的指令與 c 語言中的指令非常類似。

IO 指令：`print`、`printf()`、`getline` .....

流程控制指令：`if (...)` `{...}` `else {...}`、`while(...){...}` .....

在 awk 程式的流程中先判斷 Pattern 的結果，若為真(True)，則執行相對應的 Actions；若為假 False 則不執行相對的 Actions。若是處理的過程中沒有 Pattern，awk 會無條件的去執行 Actions。

2.2.3 工作流程：執行 awk 時, 它會反複進行下列四步驟。

- ① 自動從指定的資料檔中讀取一筆資料列。
- ② 自動更新(Update)相關的內建變數之值。
- ③ 逐次執行程式中所有的 Pattern { Actions } 指令。
- ④ 當執行完程式中所有 Pattern { Actions } 時，若資料檔中還有未讀取的料，則反覆執行步驟 1 到步驟 4。

awk 會自動重覆進行上述的四個步驟，所以使用者不需在程式中寫這個迴圈。

### 2.3 端點到端點的延遲(End-to-End Delay)。(光碟中的 Exp3\measure-delay.awk)

```
# 這是測量 CBR 封包端點到端點間延遲時間的 awk 程式
BEGIN {
# 程式初始化，設定一變數以記錄目前最高處理封包的 ID。
    highest_packet_id = 0;
}
{
    action = $1;
    time = $2;
    from = $3;
    to = $4;
    type = $5;
    pktsize = $6;
    flow_id = $8;
    src = $9;
    dst = $10;
    seq_no = $11;
    packet_id = $12;
# 記錄目前最高的 packet ID
    if ( packet_id > highest_packet_id )
        highest_packet_id = packet_id;
# 記錄封包的傳送時間
    if ( start_time[packet_id] == 0 )
        start_time[packet_id] = time;
# 記錄 CBR (flow_id=2) 的接收時間
    if ( flow_id == 2 && action != "d" ) {
        if ( action == "r" ) {
            end_time[packet_id] = time;
        }
    }
}
```

```
    }
  } else {
#把不是flow_id=2的封包或者是flow_id=2但此封包被drop的時間設為-1
    end_time[packet_id] = -1;
  }
}
END {
#當資料列全部讀取完後，開始計算有效封包的端點到端點延遲時間
  for ( packet_id = 0; packet_id <= highest_packet_id;\
    packet_id++ ) {
    start = start_time[packet_id];
    end = end_time[packet_id];
    packet_duration = end - start;
#只把接收時間大於傳送時間的記錄列出來
    if ( start < end ) printf("%f %f\n", start,
      packet_duration);
  }
}
```

■ 執行方法。(\$ 為 cygwin shell 的提示符號)

```
$awk -f measure-delay.awk out.tr
```

若是要把結果存到檔案，可使用導向的方式，把結果存到cbr\_delay檔案中。

```
$awk -f measure-delay.awk out.tr > cbr_delay
```

## ■ 執行結果：

```
0.100000 0.038706
0.108000 0.038706
0.116000 0.038706
0.124000 0.038706
0.132000 0.038706
.....
```

## 2.4 抖動率(jitter)。(光碟中的 Exp3\measure-jitter.awk)

抖動率就是延遲時間變化量(delay variance)，由於網路的流量隨時都在變化，當流量大的時候，許多封包就必需在節點的佇列中等待被傳送，因此每個封包從傳送端到目的地端的時間也就不一定會相同，而這個不同的差異就是所謂的抖動率。抖動率越大，則表示網路越不穩定。

```
# 這是測量 CBR 封包 jitter 的 awk 程式
# jitter = ((recvtime(j) - sendtime(j)) - (recvtime(i) -
# sendtime(i))) / (j - i), 其中 j > i
BEGIN {
# 程式初始化，設定一變數以記錄目前最高處理封包的 ID。
    highest_packet_id = 0;
}
{
    action = $1;
    time = $2;
    from = $3;
    to = $4;
    type = $5;
    pktsize = $6;
    flow_id = $8;
```

```
src = $9;
dst = $10;
seq_no = $11;
packet_id = $12;
# 記錄目前最高的 packet ID
if ( packet_id > highest_packet_id ) {
    highest_packet_id = packet_id;
}
# 記錄封包的傳送時間
if ( start_time[packet_id] == 0 ) {
    # 記錄下包的 seq_no
    pkt_seqno[packet_id] = seq_no;
    start_time[packet_id] = time;
}
# 記錄 CBR (flow_id=2) 的接收時間
if ( flow_id == 2 && action != "d" ) {
    if ( action == "r" ) {
        end_time[packet_id] = time;
    }
    } else {
# 把不是 flow_id=2 的封包或者是 flow_id=2 但此封包被丟棄的時間設為 -1
    end_time[packet_id] = -1;
}
}
END {
# 初始化 jitter 計算所需變量
last_seqno = 0;
last_delay = 0;
seqno_diff = 0;
# 當資料列全部讀取完後，開始計算有效封包的端點到端點延遲時間
```

```
for ( packet_id = 0; packet_id <= highest_packet_id; \
      packet_id++ ) {
    start = start_time[packet_id];
    end = end_time[packet_id];
    packet_duration = end - start;
# 只把接收時間大於傳送時間的記錄列出來
    if ( start < end ) {
        # 得到了delay值(packet_duration)後計算jitter
        seqno_diff = pkt_seqno[packet_id] - last_seqno;
        delay_diff = packet_duration - last_delay;
        if (seqno_diff == 0) {
            jitter = 0;
        } else {
            jitter = delay_diff/seqno_diff;
        }
        printf("%f %f\n", start, jitter);
        last_seqno = pkt_seqno[packet_id];
        last_delay = packet_duration;
    }
}
```

■ 執行方法。(\$ 為 cygwin shell 的提示符號)

```
$awk -f measure-jitter.awk out.tr
```

若是要把結果存到檔案，可使用導向的方式，把結果存到 cbr\_jitter 檔案中。

```
$awk -f measure-jitter.awk out.tr > cbr_jitter
```

■ 執行結果：

```
0.100000 0.000000
0.108000 0.000000
0.116000 0.000000
.....
```

2.5 封包遺失率(packet loss)。(光碟中的 Exp3\measure-loss.awk)

```
# 這是測量 CBR 封包遺失率的 awk 程式
BEGIN {
# 程式初始化, 設定一變數記錄 packet 被 drop 的數目
  fsDrops = 0;
  numFs = 0;
}
{
  action = $1;
  time = $2;
  from = $3;
  to = $4;
  type = $5;
  pktsize = $6;
  flow_id = $8;
  src = $9;
  dst = $10;
  seq_no = $11;
  packet_id = $12;
# 統計從 n1 送出多少 packets
  if (from==1 && to==2 && action == "+")
    numFs++;
}
```

```
# 統計 flow_id 為 2, 且被 drop 的封包
if (flow_id==2 && action == "d")
    fsDrops++;
}
END {
    printf("number of packets sent:%d lost:%d\n", numFs, fsDrPs);
}
```

■ 執行方法。( \$ 為 cygwin shell 的提示符號)

```
$awk -f measure-loss.awk out.tr
```

■ 執行結果：

```
number of packets sent: 550  lost:8
```

這代表 CBR 送出了 550 個封包，但其中 8 個封包丟掉了。

## 2.6 吞吐量(throughput)。(光碟中的 Exp3\measure-throughput.awk)

```
# 這是測量 CBR 封包平均吞吐量(average throughput)的 awk 程式
BEGIN {
    init=0;
    i=0;
}
{
    action = $1;
    time = $2;
    from = $3;
    to = $4;
    type = $5;
    pktsize = $6;
```

```
flow_id = $8;
src = $9;
dst = $10;
seq_no = $11;
packet_id = $12;

if(action=="r" && from==2 && to==3 && flow_id==2) {
    pkt_byte_sum[i+1]=pkt_byte_sum[i]+ pktsize;

    if(init==0){
        start_time = time;
        init = 1;
    }

    end_time[i] = time;
    i = i+1;
}
}
END {
# 為了畫圖好看，把第一筆記錄的 throughput 設為零，以表示傳輸開始
printf("%.2f\t%.2f\n", end_time[0], 0);

for(j=1 ; j<i ; j++){
# 單位為 kbps
    th = pkt_byte_sum[j] / (end_time[j] - start_time)*8/1000;
    printf("%.2f\t%.2f\n", end_time[j], th);
}
#為了畫圖好看，把最後一筆記錄的throughput再設為零，以表示傳輸結束
printf("%.2f\t%.2f\n", end_time[i-1], 0);
}
```