樹德科技大學
資訊工程研究所碩士班

碩士論文

可支援長距離網路上高速傳輸之動態 TCP Vegas 設計

Design of Dynamic TCP Vegas to Support Long-distance High Speed Networks

研究生：蘇倍豪

指導教授：洪盟峰 博士

中華民國　九十七　年　八　月

可支援長距離網路上高速傳輸之動態 TCP Vegas 設計

Design of Dynamic TCP Vegas To Support Long-distance High Speed Networks

研究生：蘇倍豪
指導教授：洪盟峰 博士

樹德科技大學
資訊工程研究所碩士班
碩士論文

A Thesis
Submitted to
Department of Computer Science and Information Engineering
Shu-Te University of Science & Technology
In Partial Fulfillment of the Requirements
For the Degree of Master
In Computer Science and Information Engineering

August 2008
Kaohsiung, Taiwan, Republic of China
中華民國九十七年八月

可支援長距離網路上高速傳輸之動態 TCP Vegas 設計

學生：蘇倍豪　　指導教授：洪盟峰 博士

樹德科技大學資訊工程研究所碩士班

摘要

本文主要目的是以不同於目前標準 TCP 演算法(TCP New Reno)的 TCP Vegas 機制為基礎，利用量測來回傳輸時間 Round-Trip Time(RTT)數值來計算連線實際吞吐量，並比較其與預計吞吐量之間差異值與α、β兩數值之間的差異，透過差異值與其預設上、下限的關係，來評估目前網路壅塞情況，以及動態調整 TCP Vegas 的視窗大小、另外之α、β值的方式，解決標準 TCP Vegas 在長距離高傳輸頻寬網路環境中，兩端點連線之間的頻寬吞吐量過小問題。本論文中，吾人捨棄傳統 TCP 所採用以封包遺失作為調節依據的作法，改採利用 RTT 量測值的變動差異與預設控制範圍（α、β）之關係來評估目前網路壅塞程度，並籍此作為傳送端 *cwnd* 提結的機制。其次，雖然此方式可較標準 TCP 的加法遞增乘式遞減(AIMD)機制來得有效率，但所設定控制範圍參數（α、β）如為固定值容易造成 *cwnd* 的管理過於保守，此一特性將使長距離連線無法有效的充分利用高速網路頻寬。因此本文提出可適用長距離網路的動態 TCP Vegas 設計方法，可以藉由動態調整的α、β值，可以不斷地因應目前網路通道的變化，改變 *cwnd* 管理方式來提高頻寬使用率。透過模擬實驗的驗證，顯示所提方式在不同傳輸距離、不同傳輸頻寬的狀況下，在點對點吞吐量上，針對瓶頸鏈路之頻寬分別為 100Mbps、155Mbps 及 622Mbps 時，量測到本機制可改善標準 TCP Vegas 吞吐量倍率分別為 7.13、11.87 及 27.79 倍，平均改善標準 TCP Vegas 倍率為 15.597 倍，由此可證實本機制可有效善目前標準 TCP Vegas 的效能。

關鍵字：壅塞控制、高速傳輸、長距離網路、TCP Vegas、服務品質保障

Design of Dynamic TCP Vegas to Support Long-distance High Speed Networks

Student：Bei-Hao Su　　Advisor：Dr. Mong-Fong Horng

Institute of Computer Science and Information Engineering
Shu-Te University

ABSTRACT

The main purpose of this research is to propose a new scheme to improve the throughput of long-distance TCP connections. The proposed scheme is a variation of TCP Vegas. Differing from the current standard TCP algorithms (TCP New Reno), the proposed scheme uses Round-Trip Time (RTT) to derive the actual throughput of the target connection. According to the difference between the actual throughput and the desired throughput, the congestion window size is adjusted to either speed op or slow down the TCP connection. For efficient response, the strategy of window-adjustment is composed four parts to meet the requirements of four stages of a TCP connection. We improve the constant-interval approach of TCP Vegas. In the new scheme, dynamic adjustment of TCP Vegas α, β value is designated to enhance the throughput of the standard TCP Vegas. Simulation results show that in various conditions of bandwidth and distance, the proposed scheme effectively improves the throughput of standard TCP Vegas. Besides, the analysis of fairness among homogenous and heterogenous TCP connections is also presented.

Keyword: congestion control, high-speed transmission, long-distance network, TCP Vegas, quality of services

目錄

圖目錄

表目錄

一、緒論

1.1 研究背景

近年來，由於網際網路使用普及率的快速成長，如何增加網路頻寬使用效能一直是令人非常熱門的研究領域，依據「台灣網路資訊中心」(TWNIC)截至 2008 年 3 月底統計我國對外頻寬成長速度資訊指出，如圖 1-1 所示，我國對外連線頻寬已達 202,172 Mbps，較上一季增加 15,729 Mbps，成長率為 8.44%。

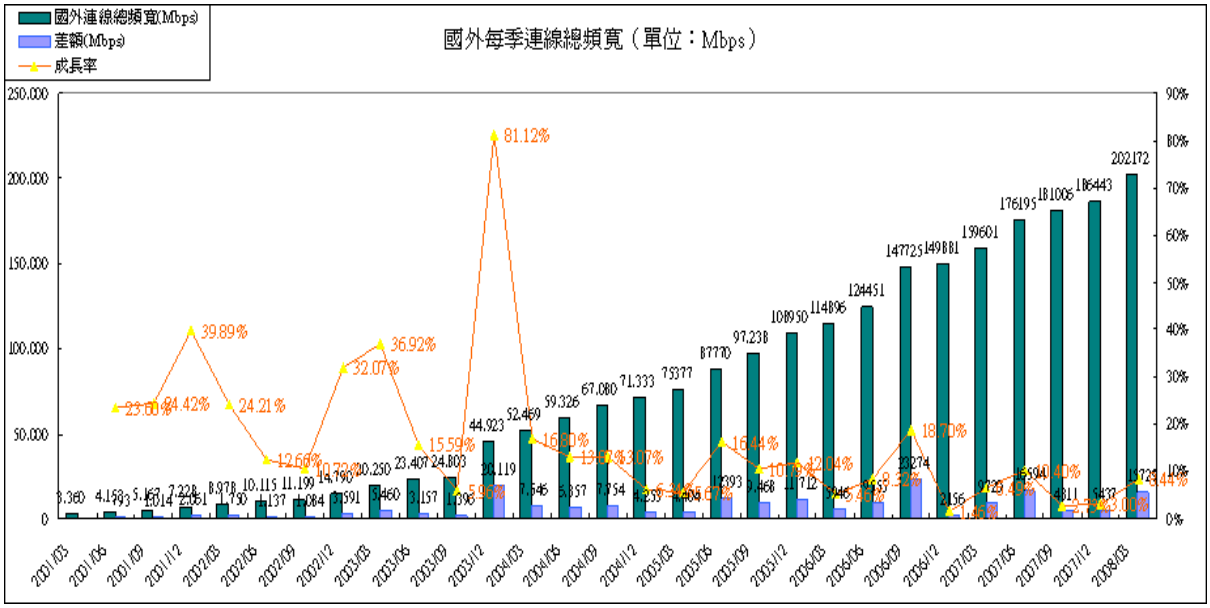

圖 1-1 我國對外每季連線總頻寬成長圖[1]

由表 1-1 可得知，我國所有對外連線頻寬中，其中連線至美國的比例佔總對外連線頻寬的 43.12%，且與上一季相比之下亦增加了 12.89%，而在連美的對外連線頻寬中，主要由ATM 155Mbps頻寬的骨幹線路為主，且有逐年增加的情況，因此由圖 1-1 與表 1-1 可知我國對網路頻寬的需求不僅是在於頻寬的提升，亦包括傳輸距離的延伸，因此可更進一步推測我國未來網路連線頻寬使用上將會不僅在短距離網路服務，在長距離高速連線服務，預計未來仍持續增加於長距離且高速傳輸類型的網路型態。

表 1-1 台灣對外連線頻寬統計表 (單位：Mbps)[1]

項目	連線國家	2007/06	2007/09	2007/12	2008/03	佔本百分比	與上季成長率
1	美國	75061	75216	77216	87168	43.12%	12.89%
2	日本	44707	43219	43841	46341	22.92%	5.70%
3	香港	30409	33653	34997	41364	20.46%	18.19%
4	中國大陸	15071	17559	17559	18804	9.30%	7.09%
5	新加坡	1625	2036	2147	2303	1.14%	7.24%
6	英國	1100	1100	2100	2100	1.04%	0.00%
7	韓國	1455	1455	1610	1766	0.87%	9.66%
8	馬來西亞	356	356	511	778	0.38%	52.08%
9	荷蘭	5599	5599	5599	622	0.31%	-88.89%
10	澳門	356	356	356	356	0.18%	0.00%
11	菲律賓	160	160	164	164	0.08%	0.00%
12	泰國	89	89	89	156	0.08%	73.82%
13	德國	89	89	89	89	0.04%	0.00%
14	澳洲	47	47	47	47	0.02%	0.00%
15	印度	45	45	45	45	0.02%	0.00%
16	紐西蘭	0	0	45	45	0.02%	0.00%
17	越南	10	10	10	10	0.01%	0.00%
18	沙烏地阿拉伯	10	10	10	10	0.01%	0.00%
19	印尼	6	6	6	6	0.00%	0.00%
	總計	176195	181006	186443	202172	100.00%	8.44%

1.2 研究動機

由於長距離且高速傳輸網路將是未來網路的趨勢之一，而此種網路包含以下特性：(1) 高線路頻寬；(2) 長傳輸距離(即高延遲 RTT)。舉例來說，台灣與美國的距離大約 9000 公里以上，中間經由數個路由器，封包的來回傳遞時間(round trip time, RTT)大約在 220ms 以上，假設此時兩端點的連線頻寬為 1Gbps 以上，則此種網路型態即符合上述長距離且高速傳輸網路的特性。

在上述長距離且高速傳輸網路的特性之下，目前的 TCP 壅塞控制法無法有效利用整個網路頻寬，主要是因為目前的 TCP 壅塞控制法(TCP New Reno)所採用的加法遞增乘式遞減(Additive Increase Multiplicative Decrease, AIMD)機制[2]，是以 封包遺失作為調整壅塞視窗(congestion window, cwnd)的依據，採用此方法調整目前網路頻寬不但會造成網路頻寬週期性的起伏，也因受長距離且高速傳輸網路的特性之下，造成調整機制對實際網路頻寬狀況的反速度不夠快速，也引起網路頻寬使用上的浪費。

因目前 TCP 壅塞控制法存在著上述缺點，故 Brakmo 和 Peterson 在 1995 年提出使用 RTT 量測來估計網路頻寬狀況進而據此調整傳送速度的壅塞控制法，稱之為 TCP Vegas[3]。TCP Vegas 藉由比較預期的吞吐量和實際的吞吐量差值(diff)以決定是否增加或減少原本 TCP 視窗的值。經由研究指出[4]，此方式較目前的 TCP New Reno 更有效率的使用網路頻寬，故本文主要採用 TCP Vegas 機制為基礎，在此機制上做修正。

雖然 TCP Vegas 較 TCP New Reno 更有效率的使用網路頻寬，但由於 TCP Vegas 是依據量測到的吞吐量差異值再與所預設的固定上限與下限比較後作為管理壅塞視窗的大小，而此方式應用在長距離且高速傳輸網路時，會因其高 RTT 值及高頻寬傳輸速率的特性，迫使目前 TCP Vegas 存在以下的問題：

1. 調整壅塞視窗的速率不夠快速，因而無法有效利用頻寬。
2. 當網路頻寬未達可用頻寬之上限且預期傳送的吞吐量與實際傳送的吞吐量差異值又座落在 α 、 β 之區間時，易造成壅塞視窗不再增減而無法有效利用整個頻寬。

1.3 研究目的

由於 TCP Vegas 在長距離高速傳輸網路下有著上一章節所提出之問題，而造成網路頻寬之浪費，因此本論文將核心著重於如何改善 TCP Vegas 連線其壅塞視窗的管理方式，以提高連線吞吐量(Throughput)，並採用動態調整α、β數值的方式取代原有 TCP Vegas 中固定的α、β數值，加以改善壅塞視窗限制不成長的情況。故本論文主要目的如以下幾點：

1. 提高長距離且高速傳輸連線之吞吐量。
2. 提高在長距離高速網路上，TCP 連線吞吐量達到瓶頸頻寬的速度。
3. 提高長距離高速網路頻寬的使用率。

1.4 文章組織

本文章節組織如下，在第二章中，將討論長距離高速傳輸網路下，現有 TCP 機制的特性與問題，接著將探討 TCP 的現有壅塞控制技術，其中包括傳統 TCP 壅塞控制法、現有新式 TCP 機制的方法。第三章提出可支援長距離網路上高速傳輸之動態 TCP Vegas 設計的架構，其機制主要都是透過 diff 的量測與α、β作比較後，改變原來 TCP Vegas 機制增減擁視窗之數值，並提出 TCP Vegas 視窗上限與下限值管理的方式有效增減α、β數值取代原本固定之α、β數值，提出改善在長距離網路下高速傳輸的吞吐量。第四章中，使用模擬工具 NS2 進行網路模擬實驗，依不同的網路拓樸情境，分別作不同的實驗，並從一連串的實驗環境，將分析模擬過後的數據，以作為往後研究的參考。最後第五章為結論與未來展望，將本文所提出的方法做重點式整理，並逐一簡單分析與討論，最後提出未來可供研究的方向。

二、文獻探討

2.1 長距離高速傳輸網路特性分析

為了改善長距離高速傳輸網路之下的 TCP 連線效能，了解目前 TCP 機制在此種網路特性下之效能是必須的，因此本小節將以長距離高速傳輸網路的特性，即長距離(高延遲往返時間)與高傳輸頻寬對 TCP 機制的網路效能影響分別做一說明。

1. 高延遲往返時間之影響

由於 TCP 採用可靠傳輸的方式，如圖 2-1 所示，當 A 端，也就是發送端發送 Packet 1 至 Packet3 給 B 端即接收端之後，需等到 B 端回應 ACK1 至 ACK3 (acknowledgement, ACK)給 A 端之後，A 端才會繼續發送 Packet4 至 Packet6 給 B 端；從 A 端發送 Packet 1 至收到 B 端 ACK1 回來的這段時間即為此次的 Round-Trip Time (RTT)。

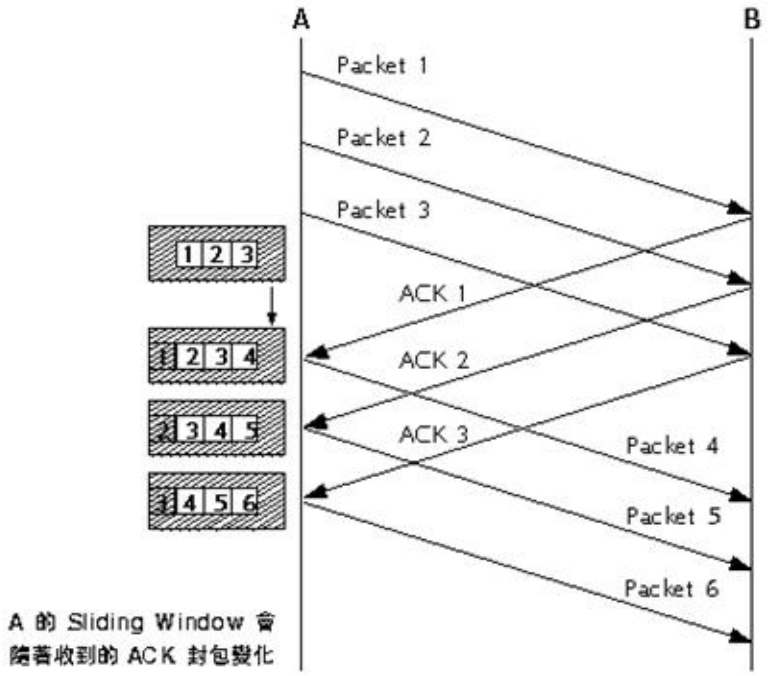

圖 2-1 TCP 封包傳送圖[5]

TCP 所採用的可靠傳輸方式應用在短距離網路上，也即是在低延遲往返時間時，不論是採用封包遺失作為調整壅塞視窗依據的 TCP New Reno 機制或是使用量測 RTT 的方式作為調整壅塞視窗依據的 TCP Vegas 機制，均可快速的調整壅塞視窗的大小以因應實際網路情況，如圖 2-2(以 TCP Vegas 機制為例)，在 RTT 為 24ms 之下，壅塞視窗調整速度快速，大約 4 秒時將吞吐量調整至穩態的狀況。

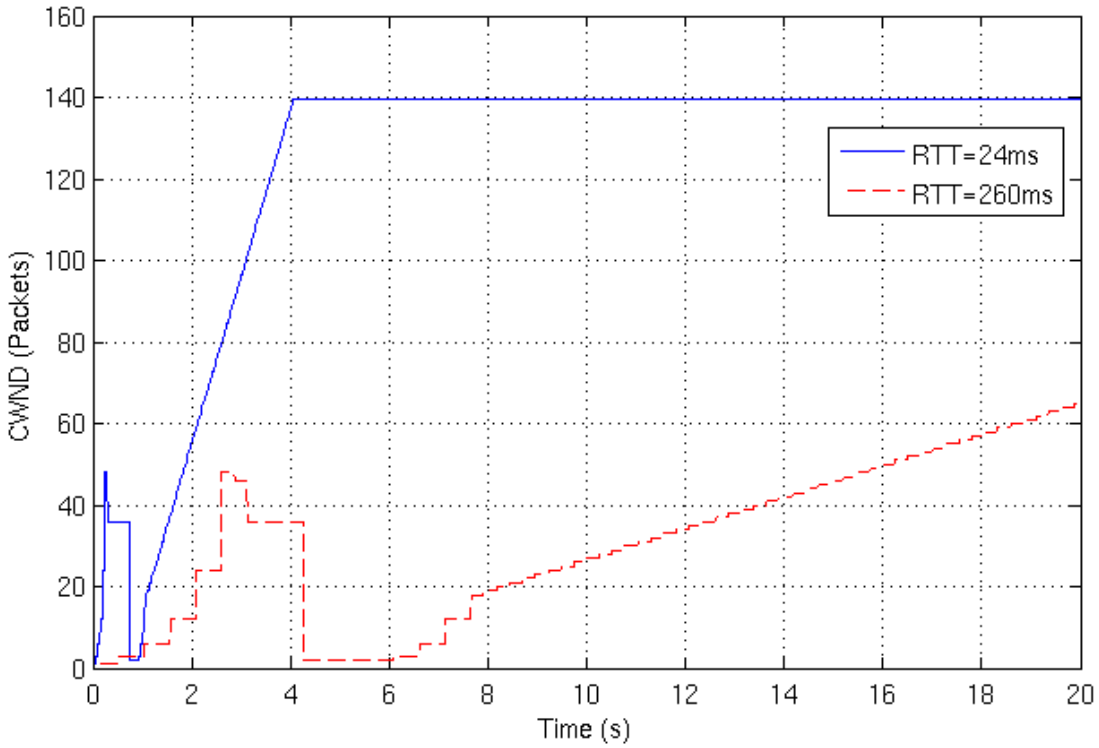

圖 2-2 高延遲與低延遲往返時間之 TCP Vegas cwnd 成長圖

反之當 TCP 連線只有高延遲往返時間時，將會因為下一個封包等待新一個封包的確認信號過久，使得 TCP 機制在調整壅塞視窗並無法有效的對應到實際網路情況，造成 TCP 機制在調整壅塞視窗的效率上，產生了嚴重的鈍化影響，也因此造成 TCP 機制在高延遲往返時間的情況之下網路吞吐量不佳，如圖 2-2 中當 RTT 增加至 260ms 時，TCP 機制在壅塞視窗調整上由於 RTT 的增加，而且過早進入壅塞避免階段，調整速度呈現出緩慢的情況，雖然與低延遲往返時間一樣有著相同的 45Mbps 網路頻寬，卻因 RTT 的變大，而造成

壅塞視窗遲遲未達網路頻寬上限，在模擬時間 20 秒時，傳輸吞吐量僅能達到 1.49Mbps，大約只達可用頻寬上限的三十分之一，故目前 TCP 機制並不適用於高延遲往返時間網路。

2. 網路傳輸頻寬高低之影響

如圖 2-3 所示，當 TCP 機制應用在低傳輸頻寬時，可快速的達到頻寬上限，顯示出 TCP 機制足以應付低傳輸頻寬的網路情境，但是當網路傳輸頻寬上限增加之後，TCP 機制達到高傳輸頻寬上限之時間將也隨之大幅度增加，如圖 2-3 所示(以 TCP Vegas 機制為例)，當網路傳輸頻寬上限由原本 45Mbps 增加至 622Mbps 後，TCP 機制則由原本只需花費 3 秒的時間即可達到網路傳輸頻寬之上限，增加至需要花費 42 秒內的時間才可達到 622Mbps 的網路傳輸頻寬上限，由此可知 TCP 機制應用在高傳輸頻寬時，壅塞視窗的成長速度尚不足以應付此種網路情境。

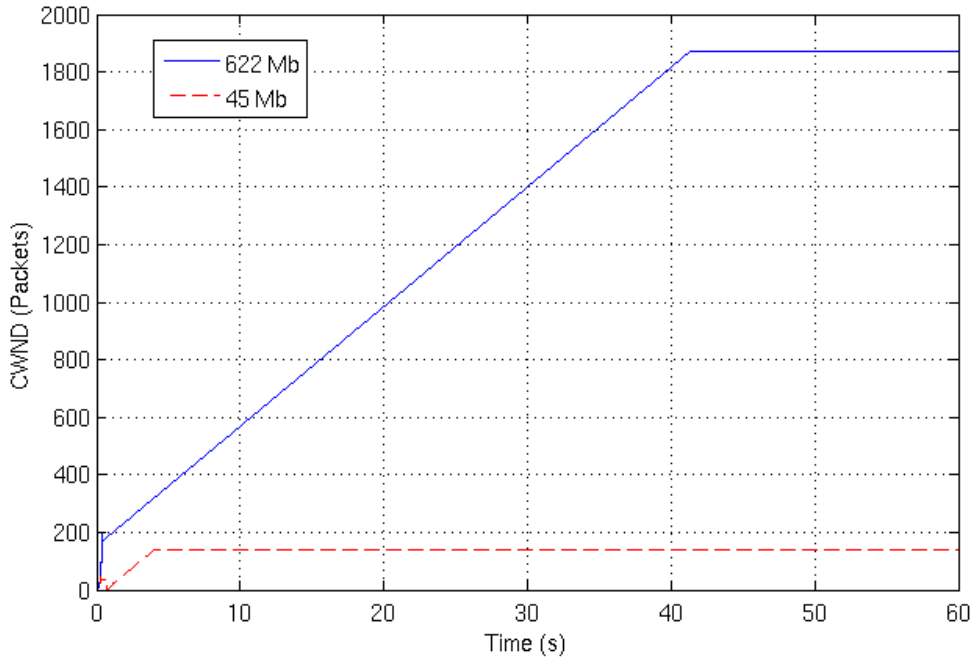

圖 2-3 高傳輸頻寬與低傳輸頻寬之下 TCP Vegas cwnd 成長圖

總合上述高延遲往返時間與頻寬高低兩點分析可知，TCP 機制目前應用在長距離高速傳輸頻寬網路上時，至少有著因高延遲往返時間造成機制調整壅塞視窗之鈍化及高傳輸頻寬之下而突顯出機制調整速度不及網路狀況變化速度的問題。

2.2 現有 TCP 壅塞控制機制之研究

目前研究改進 TCP 壅塞控制機制的方式多不勝數，其中現在最廣泛使用的方式，首推 TCP New Reno，故本小節將分析目前最廣泛使用以封包遺失為調整壅塞視窗依據的 TCP New Reno 機制的優缺點，之後再分析採用以量測 RTT 為調整壅塞視窗依據的 TCP Vegas 機制，並比較此兩種機制的優缺點，以此解釋為何本文採用 TCP Vegas 機制而不採用目前 TCP New Reno 機制為本文主要的基礎機制，之後再於下一節中提出目前改進 TCP Vegas 機制的作法。

2.2.1 TCP New Reno 機制

TCP New Reno 連線的生命週期可以分為兩個階段，分別是緩起動階段(slow-start)與壅塞避免階段(congestion avoidance)，基本上由兩種主要演算法構成，分別是緩起動演算法與壅塞避免演算法，藉由緩啟動門檻值(Slow-Start Threshold, ssthresh)決定何時採用何項演算法。詳細機制如下：

在緩啟動階段，發送端每次接收到 ACK 時，即增加視窗大小，而增加方式為指數方式遞增，也就是發送端接收到一個 ACK 時，下一個時間發送端增加一個 *cwnd* 大小送出，*cwnd* 成長方式如式(1)所示。

$$cwnd(k+1) = cwnd(k) + 1 \tag{1}$$

當過了緩啟動門檻值後，即進入壅塞避免階段，在此階段 *cwnd* 的成長速度由指數方式遞增改為線性增加，也就是發送端接收到了等同 *cwnd* 個 ACK 時，下一個時間發送端才會增加一個 *cwnd* 大小送出，此時 *cwnd* 更新方式如式(2)所示。

$$cwnd(k+1) = cwnd(k) + \frac{1}{cwnd(k)} \tag{2}$$

另一方面，若發送端沒有收到 ACK 時，那麼在下一個時間發送端傳送的 *cwnd* 大小將減少一半，如式(3)所示，並把緩啟動門檻值減為目前 *cwnd* 的一半。

$$cwnd(k+1) = cwnd(k) - \frac{cwnd(k)}{2}$$
$$ssthresh = \frac{cwnd(k)}{2} \tag{3}$$

隨即發送端將進入快速重傳(Fast Retransmission, FR)的階段，將遺失的封包快速重送給接收端，等發送端收到遺失封包的 ACK 之後，再重回到壅塞避免階段，採用線性增加方式遞增，直到又發生封包遺失，如此不斷地重複以上的演算機制。因此，TCP New Reno 的壅塞控制方式會使受控連線的吞吐量呈現週期性的起伏。

大致上 TCP New Reno 機制所引起的 TCP 連線吞吐量變化將如圖 2-4 所示，從一開始緩啟動階段，以指數增加的方式增加 *cwnd*，在 *cwnd* 為 24 時通過緩啟動門檻值而進入壅塞避免階段，發生封包遺失後將 *cwnd* 減為一半，進入快速重傳階段，等重傳全部遺失的封包後，再採用線性增加的方式增加 *cwnd*，如此不停的重複以上情況。

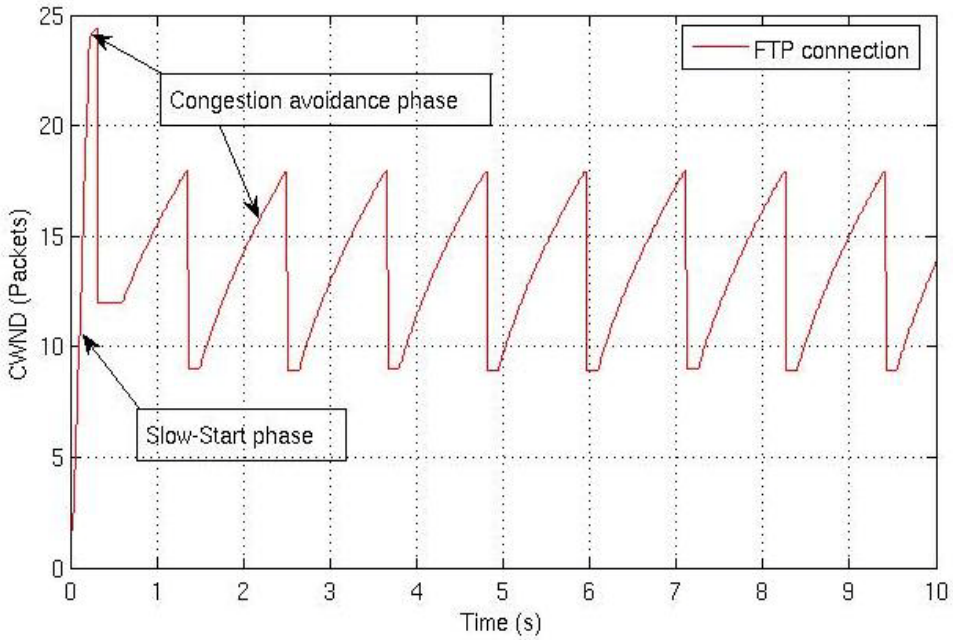

圖 2-4 TCP New Reno 機制圖

TCP New Reno 機制應用在低頻寬、短距離網路上可達到公平且不錯的網路使用效率，但應用在長距離高傳輸頻寬網路時，將會因高延遲往返時間引起調節速度緩慢而造成網路效能不佳。主要原因是由於當它進入壅塞避免階段後，壅塞視窗成長速度過於緩慢，將會無法有效利用整個高傳輸頻寬，而且封包一旦發生遺失，由減半的壅塞視窗回復到最高傳輸頻寬所需花費的時間過久，故在這兩種時間點內皆無法有效利用整個網路頻寬，且由圖 2-4 可看出，後者封包遺失的情況將會以週期性的方式出現，也即是無法有效利用網路頻寬的情況將會不斷地以週期性的方式出現；另外由於 TCP New Reno 機制採用封包遺失作為調整壅塞視窗的依據，假設封包已發生遺失但發送端尚未得知的這段期間內，將會造成發送端依然持續發送封包，而造成更多的封包遺失，造成網路頻寬使用上的浪費。

2.2.2 TCP Vegas 機制

TCP Vegas 機制捨棄了 TCP New Reno 機制採用封包遺失作為調整壅塞視窗準則的方式，改採用量測每回合 RTT 的方式進行調整壅塞視窗大小，以此方式改善 TCP New Reno 機制週期性封包遺失的缺點，故本節將介紹 TCP Vegas 機制，並探討其在長距離高傳輸頻寬網路上的特性。

TCP Vegas 機制希望除了能有效地使用可用頻寬，也要防止傳送太快而發生封包遺失的情況發生，故對原本的 TCP New Reno 機制上，TCP Vegas 機制修改了緩啟動演算法、壅塞避免演算法及快速重傳演算法。

TCP Vegas 機制在緩啟動的階段，將原本收到一個 ACK 即將 *cwnd* 增加一個的方式，修改為經過兩個 RTT 時間之後才會增加一倍 *cwnd*。另外 TCP Vegas 機制採用與 TCP New Reno 機制不同的方式計算緩啟動門檻值，TCP Vegas 機制根據預期傳送率和實際傳送率之間的差異值來調整緩啟動門檻值，當 TCP Vegas 機制偵測到網路開始有佇列產生時，TCP Vegas 機制即由緩啟動階段進入壅塞避免階段。

不同於 TCP New Reno 機制的壅塞控制演算法，TCP Vegas 壅塞控制演算法藉由觀察 RTT 的變化來控制 *cwnd* 的大小，其基本演算法大致如式(4)(5)：

$$\begin{aligned} diff(k) &= (Expected - Actual) * BaseRTT \\ &= \left(\frac{cwnd(k)}{BaseRTT} - \frac{cwnd(k)}{RTT(k)} \right) * BaseRTT \end{aligned} \tag{4}$$

$$cwnd(k+1) = \begin{cases} cwnd(k)+1, diff(k) < \alpha \\ cwnd(k), \alpha < diff(k) < \beta \\ cwnd(k)-1, diff(k) > \beta \end{cases} \tag{5}$$

TCP Vegas 機制首先定義了 *BaseRTT* 與 *RTT* 兩個數值，分別為目前量測到的最小 RTT 數值與此次量測到的 RTT 數值。再由式(4)定義了此回合的差值 *diff*，其中 *Expected* 指的是預期傳送的吞吐量，*Actual* 則指的是實際傳送的吞吐量；前者是由目前的 *cwnd* 除以 *BaseRTT* 求得，而後者則是由此目前的 *cwnd* 除以目前的 *RTT* 求得；在求得 *diff* 值的變化之後，再依據式(5)作下一回合 *cwnd* 大小的調整，α、β 預設數值分別為 1 與 3。

由式(5)可知，當 *diff* 值大於 β 值時，代表傳送速率太快，因此應該減低 *cwnd* 以減緩傳送的速率，故在下一回合 *cwnd* 傳送量將減少一個，反之當 *diff* 值小於 α 值時，代表傳送速率太慢，因此應該增加 *cwnd* 以提高傳送的速率，則下一回合 *cwnd*

傳送量則增加一個；當 *diff* 值介於α、β之間，則維持目前網路速度，故 *cwnd* 在下一回合並不做增減的動作。TCP Vegas 所管理的連線，其視窗值變化情況如圖 2-5 所示，由一開始緩啟動階段不斷提升 *cwnd*，在 0.5 秒時離開緩啟動階段而進入壅塞避免階段，大約在 0.6 秒時即進入穩態而不再增減 *cwnd*。

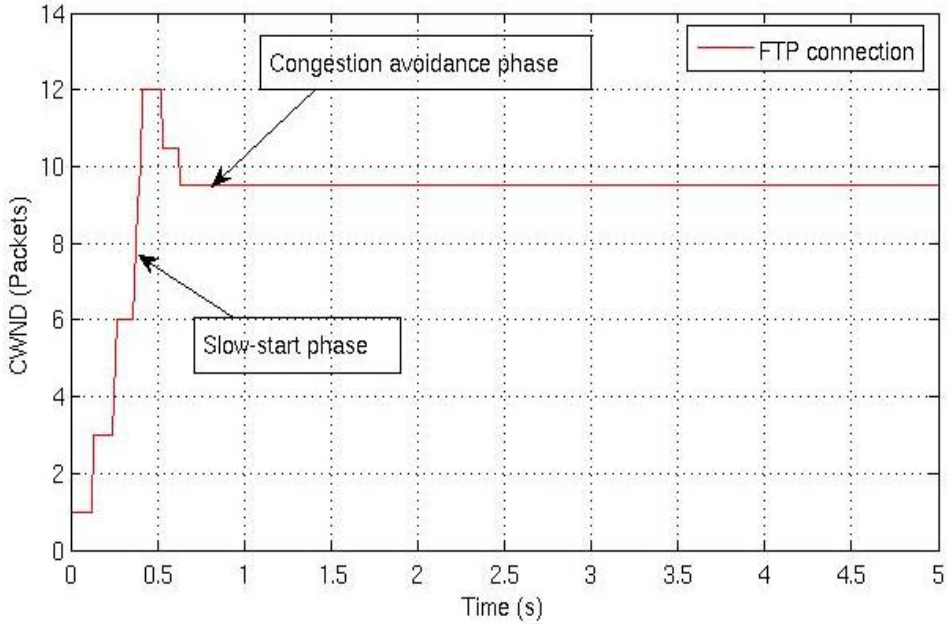

圖 2-5 TCP Vegas 機制圖

由圖 2-5 可看出，TCP Vegas 機制改善 TCP New Reno 機制原有週期性封包遺失的問題，且因為 TCP Vegas 機制採用量測每回合 RTT 的方式調整 *cwnd*，比起 TCP New Reno 機制採用封包遺失作為調整更能快速的因應實際網路情況。

雖然 TCP Vegas 機制改善 TCP New Reno 機制的缺點，但應用在長距離網路情境時仍會碰到以下幾點問題：(1) 因 RTT 值過大而造成 *cwnd* 更新速度過慢而無法即時因應整個網路頻寬；(2) 若網路尚有可用頻寬且當 *diff* 值介於α與β之間時，整個網路頻寬將不再增減，此時則會發生無法完整利用頻寬的情況；(3) *cwnd* 成長速度仍不足夠應付高傳輸頻寬網路。

2.3 現有改進 TCP 壅塞控制機制之研究

目前許多研究者會在 TCP 各種參數上作些許變動，以期望網路的效能達到最

佳狀態，如[6]、[7]和[8]為例。不過這些研究應用在不同網路情境時，也許就不能適用，因此這類型的研究，大都只適合個案，並不適合大眾使用的實際網路，也因此這些改進 TCP 機制的方式在本節中並不作探討。

此外，經由上一節分析出目前 TCP New Reno 機制與 TCP Vegas 機制的優缺點比較之後，故在此節將舉出目前修正 TCP 機制的一些方式，以下列兩點作分類：(1)依據封包遺失調整壅塞視窗；(2)依據量測 RTT 調整壅塞視窗，並分析此兩類的優缺點。

2.3.1 依據封包遺失為調整壅塞視窗基礎之改進方式

有許多研究者投入在修改目前 TCP New Reno 壅塞控制，而這邊將探討在長距離高速傳輸網路環境下，修改目前 TCP New Reno 壅塞控制，來改善網路整體效能。其中以 HighSpeed TCP[9]、FAST TCP[10]、Scalable TCP[11]與 H-TCP[12]為有名，而本文只分析探討 HighSpeed TCP、FAST TCP 與 Scalable TCP 為主要。

I. HighSpeed TCP

HighSpeed TCP主要應為在長距離高頻寬傳輸網路上，有效改善資料的傳輸速率。在HighSpeed TCP機制中主要使用三項參數作為衡量基準，分別為W_L、W_H和P_H三個參數。HighSpeed為了能與TCP New Reno共存，所以特別設定了W_L參數，若是*cwnd*小於W_L值時，則壅塞控制的機制將會與TCP New Reno相同，相反地，若是*cwnd*大於W_L值時，那麼機制將會以HighSpeed所提出的機制運行，如式(6)(7)所示。

$$cwnd(k+1) = cwnd(k) + \frac{a(cwnd(k))}{cwnd(k)} \tag{6}$$

$$cwnd(k+1) = cwnd(k) - b(cwnd(k)) \times cwnd(k) \tag{7}$$

式子中的(6)(7)可從式(9)(10)(11)得到變數值。在式(9)中，為發送端接收到上一個時間所送出 *cwnd* 數量的 ACK 時，發送端再下一個時間所增加的 *cwnd* 數目。而式(10)則是發送端一旦少收到原本應該收到的 ACK 數目時，那麼發送端在下一個時間將會減少多少的 *cwnd* 數量，主要目的是用來減緩發送

端傳送速度過快的問題。最後式(11)中，以 log 方式，搭配封包遺失率，算出目前網路大概可容納的 *cwnd* 數量，以提供式(9)(10)作精確運算，不但可控制發送端在不同時間的傳輸速率，同時還能藉由式(11)持續算出網路目前還可容納多少 *cwnd* 或是網路頻寬已達到飽和，需要降緩發送端的傳送速率，以避免封包發生遺失。

$$a(cwnd(k)) = \frac{2*(cwnd(k))^2 * b(cwnd(k)) \times p(cwnd(k))}{2 - b(cwnd(k))} \quad (9)$$

$$b(cwnd(k)) = \frac{\log(cwnd(k)) - \log(W_{low})}{\log(W_{high}) - \log(W_{low})}(b_{high} - 0.5) + 0.5 \quad (10)$$

$$p(cwnd(k)) = \exp[\frac{\log(cwnd(k)) - \log(W_{low})}{\log(W_{high}) - \log(W_{low})} \times \{\log(P_{high}) - \log(P_{low})\} + \log(P_{low})] \quad (11)$$

綜合以上，可以看出 HighSpeed TCP 擁有一些優點，如高吞吐量、TCP 的友善性，以及回應速度。雖然 HighSpeed TCP 能夠從網路上得到不錯的效能，但是在現實的網路環境中，HighSpeed TCP 還是得先設定必要參數才可運作，偏偏又由式(9)(10)(11)可看出，要如何設定出完善的參數值是一門大學問，且設定又如此複雜，因此要應用在實際網路上是有困難的。

II. Fast TCP

Fast TCP 的演算法，主要解決在高遺失率的網路環境，而在高速網路環境中，若是常常發生遺失時，可能會浪費許多時間等待接收端傳回 ACK，由於距離過長，更容易使得發送端作傳送控制過於遲鈍，無法快速依據網路變化，而改變發送端的傳送速率，導致封包容易不斷發生遺失。因此，Fast TCP 做法也是透過封包遺失率與佇列延遲方式，在可能容易發生遺失率的長距離高速傳輸網路上，提供發送端在不同 RTT 時間，作傳輸速度改變，因此提出式(12)，藉以改善長距離高速傳輸網路上，若是在不斷發生封包遺失時，如何調整發送端的傳送速率。

$$cwnd(k+1) = \min\{2cwnd(k), (1-\gamma)cwnd(k) + \gamma(\frac{baseRTT}{RTT(k)})cwnd(k) + \alpha(cwnd(k), qdelay)\} \quad (12)$$

上述方程式的 $\gamma \in (0,1]$，*baseRTT* 為最小 RTT 時間，另外還採用移動加權平均值方式，以及 α 值與 *cwnd* 和 *qdelay* 作為估計發送端再下一個時間，調整傳送速率的方法，而這裡的 α 值為固定參數，另外 *qdelay* 則是端點對端點的平均佇列延遲。經由此方程式，由 RTT 與 *qdelay* 方法作為發送端再此一連線時，調整資料傳送的速率的主要依據。若是長距離高速傳輸網路環境上，若是連線數不多時，那麼發送端在提升傳送速率顯然沒有優於其他學者提出來方式，主要原因為 Fast TCP 過於謹慎調整發送端的傳送速率，若是在長距離網路上，那麼無法因應網路的變化，由於距離過長，發送端得經由很長一段時間，才能改變傳送速率，因此，Fast TCP 還是在長距離高速傳輸網路上，還是過於遲緩改變發送端的傳送速率。

III. Scalable TCP

Scalable TCP 是以修改 TCP New Reno 中的 AIMD 機制，在長距離高速傳輸網路環境上，讓每條連線，能夠得到最好的吞吐量。利用 TCP New Reno 的 AIMD 機制中，發送端收到 *cwnd* 個 ACK 時，才增加一個 *cwnd* 大小，修改為發送端每收到一個 ACK 時，發送端在下一個時間隨即增加 0.01 個 *cwnd*，換句話說也就是當發送端收到一百個 ACK 時，發送端再下一個時間時，則會遞增一個 *cwnd* 大小，如式(13)所示。另外若原先 TCP New Reno 封包發生遺失時，發送端在下一次時間，隨即減少二分之一的 *cwnd* 數量，而 Scalable TCP 方法則是修改為減少到原來 *cwnd* 值的四分之三，也就是發生封包遺失時，讓 *cwnd* 比原先 TCP New Reno 機制減少數量較之前的數值少，以便於能再次快速進入 AI 的遞增方式，提升發送端的傳送速率，如式(14)。透過這樣的參數修改，在每一條連線，皆能快速提升在長距離高速傳輸網路的吞吐量。

$$cwnd(k+1) = cwnd(k) + 0.01 \quad (13)$$

$$cwnd(k+1) = cwnd(k) - 0.125 * cwnd(k) \quad (14)$$

雖然使用 Scalable TCP 機制可以大幅改善發送端在每次時間傳送 *cwnd* 太少的問題，而且在長距離高速傳輸網路環境下，更是如此，且在多條連線之下，此機制的方式還是容易造成封包遺失，由於 Scalable TCP 機制僅在增加與遞減上作參數上的變化，由於在網路上，能夠得到的資訊是 RTT，因此，Scalable TCP 如何在多條連線下，還能保持每一連線高吞吐量，並且能隨時控制發送端的傳送速率，而不是等到每次發生遺失時，才作傳送端速度的改變，因此，無法有效保持每一條連線的高吞吐量，這將會使 Scalable TCP 面臨與 TCP New Reno 一樣的致命缺點。

2.3.2 依據量測 RTT 為調整壅塞視窗基礎之改進方式

I. Enhanced Vegas

在 TCP Vegas 機制中，RTT 是調整 *cwnd* 大小一個重要依據，然而目前網路傳輸上，RTT 卻一直受到正向傳輸的壅塞或反向傳輸的壅塞影響 TCP Vegas 調整 *cwnd* 大小。理論上因反向傳輸而造成的壅塞並不會影響到正向的傳輸，故當反向流量壅塞發生的時候，並不需要降低封包的傳輸率，然而實際上卻會因為反向傳輸的壅塞而造成正向傳輸減少傳輸，因而造成網路頻寬無法有效利用。

在 Enhanced Vegas[13]此論文中，測量到的 RTT 由原本發送封包時間到收到 ACK 這段時間，分割成以下四個部分：

1. 正向的固定延遲時間(forward fixed delay time)
2. 正向的等待時間(forward queuing time)
3. 反向的固定延遲時間(backward fixed delay time)
4. 反向的等待時間(backward queuing time)

固定延遲時間定義為封包傳送的時間與封包處理時間的總和；等待時間則定義為封包在佇列等待被傳送的時間。

假設當反向傳輸的壅塞產生時，反向等待時間增加，導致實際吞吐量減少。如此一來預估吞吐量和實際吞吐量的 *diff* 差異值變大，TCP Vegas 機制將

會減少 *cwnd*。

因此為了更有效地利用網路頻寬，Enhanced Vegas[5]修改了 TCP Vegas 測量實際吞吐量的方法。它將實際吞吐量定義如式(15)所示：

$$Actual'(k) = \frac{cwnd(k)}{RTT_{new}(k) - QD(backward)} \qquad (15)$$

RTTnew 表示目前所測量到的 RTT 值。*cwnd* 則表示目前的壅塞視窗大小。*QD(backward)*則表示反向的等待時間。在公式(15)中，可以看出 *Actual'* 即為此回合去掉反向等待時間所造成的延遲後所能達到的吞吐量。

在此方法中，必須利用 TCP timestamps option 來計算正向和反向的封包等待時間。來源端在傳送封包時，會將 timestamp 加入到 TCP header 裡，當目的端收到封包後，正向的 timestamp 會被記錄起來，並加入反向的 timestamp 到 ACK 封包裡。

依此論文的實驗結果，得知以 *Actual'* 代替 TCP Vegas 機制的 *Actual*，計算較符合網路狀況的封包傳輸率，便能排除反向壅塞造成封包傳輸率不正確調整的影響，且能有效的提高 TCP Vegas 機制網路效能的使用。

雖然此機制可有效改善因反向傳輸壅塞而造成正向傳輸使用網路效能的不佳，但由於需要在 TCP 標頭上多新增欄位以記錄正向及反向時間，在實用性上並不高，故只適合應用在模擬環境上使用，且此機制欠缺提高調整 *cwnd* 大小的機制，應用在高頻寬傳輸網路時將無法有效的利用網路頻寬。

II. STT-Vegas

STT-Vegas[14]為另一種利用排除 RTT 反向傳輸影響調整封包傳輸率不必要因素的 TCP Vegas 修改版本。STT-Vegas 提出了更簡單的 RTT 計算方式，和 Enhanced Vegas 一樣，利用在 TCP header 裡加 timestamp 來計算 RTT。首先將 RTT 分為正向 *F_STT* 和反向 *B_STT* 總和的 *STT* (Single-Trip Time)，在這裡 *STT* 被定義為一個封包產生於來源端和目的端接收到封包的時間差，*F_STT* 即為

發送端封包傳輸至接收端所花費之時間，反之 *B_STT* 即為接收端回送 ACK 回發送端這段時間。因此 *STT* 如式(16)示：

$$STT(k) = F\_STT(k) + B\_STT(k) \tag{16}$$

在每一回合的 STT，當封包被送達目的端時，將 *F_STT* 紀錄在接收端的 ACK 標頭裡，並由 ACK 回送給發送端。將目前量測到最小數值的 STT 定為 *BaseSTT*，同樣地 *BaseSTT* 也可表示如式(17)：

$$BaseSTT(k) = F\_BaseSTT(k) + B\_BaseSTT(k) \tag{17}$$

利用式(8)，從最小的 *BaseSTT* 扣除最小的數值的正向傳輸時間(*F_BaseSTT*)即可計算出反向傳輸所需花費的最小時間(*B-BaseSTT*)數值。為了排除在反向傳輸壅塞對正向傳輸不必要的影響，所計算出新的 *RTT'* 如式(18)所示：

$$RTT'(k) = F\_STT(k) + B\_BaseSTT(k) \tag{18}$$

將新的 *BaseSTT* 與 *RTT'* 兩參數分別取代原來計算預期傳送的吞吐量中 *BaseRTT* 參數與實際傳送的吞吐量中 *RTT* 參數，即成為 STT-Vegas 主要的機制。

此機制雖然較 Enhanced Vegas 演算法簡單，可是也同樣的修正了 TCP Vegas 機制下量測 RTT 的缺失，但由於需要將現有的 TCP 標頭新增欄位記錄單向的傳輸時間，故實際應用在目前網際網路上仍是無法實現，且應用在高傳輸頻寬網路時，因其並未提高調整 *cwnd* 大小，故將無法有效地發揮整個網路效能。

III. Ada Vegas

在 Ada Vegas [15]中，同樣與 TCP Vegas 機制一樣利用預期傳送吞吐量與實際傳送吞吐量的差值 *diff* 與 α、β 這二個上、下限的比較來調整壅塞視窗。在此機制中加入了一個動態控制壅塞視窗成長倍數的變數 *inc*，在 TCP Vegas

機制中 *cwnd* 成長數值固定設為 1，在此機制中即表示為 *1* MSS*(Maximum Segment Size)。TCP Vegas 機制在 *diff* < α 時，是以 *1*MSS* 來增加 *cwnd* 數值，而 Ada Vegas 機制則改以 *inc*MSS* 代替，*inc* 則依據下面的規則動態調整。首先 Ada Vegas 機制另外記錄在 diff < α 時，連續進入此區間的次數 *succ*。接著當 *diff* < α 時，Ada Vegas 機制將 α 、β 及 *inc* 參數調整成相對應的數值，如表 2-1 所示。

表 2-1 TCP Ada Vegas 機制調整動態參數表

	α	β	*inc*
0 ≦ *succ* ≦ 3	1	3	1
4 ≦ *succ* ≦ 7	2	4	2
8 ≦ *succ* ≦ 15	2	4	4
16 ≦ *succ*	4	8	8

隨著 *inc* 提高，封包傳輸率也會將對地提高，在此機制中，*inc* 採用倍數增加，而隨著 *inc* 的提高之外，α 、β 參數也隨著提高，如此有可能會發生一次將封包傳輸率調整過高而造成封包壅塞的情形，因此 Ada Vegas 機制在減少封包傳送率方面也做了些調整。

Ada Vegas 機制新增一參數 *lastInc*，用以記錄封包傳送的最大倍數值，當 *diff*< α 時，*lastInc* 會被設為 *inc* 的數值。而假設經過二次的 RTT 時間後，若狀況仍維持在 $\alpha \leqq diff \leqq \beta$ 的穩定狀態，這時將會把 *lastInc* 重設為 1。等到網路壅塞情形發生時，即 β <*diff*，Ada Vegas 機制會將 *lastInc* 設為 *max(lastInc*/*2 , 1)*，並利用此值來減少 *cwnd* 數值。

Ada Vegas 機制對於壅塞避免機制對於網路頻寬的使用效率雖然大大地提升了許多，可是應用在長距離高速傳輸網路上仍是不足，主要原因在於其 *cwnd* 成長數值仍是太過保守，應用在高頻寬網路上無法有效的提升網路吞吐量至頻寬上限，但其調整 α 、β 數值概念卻因此被當成新的調整 *cwnd* 依據之一，如 TCP Vegas-A [16]也即是以此概念所提出改善 TCP Vegas 機制效能的方式之一。

IV. Quick Vegas

在 Quick Vegas[17]機制中主要修正 *cwnd* 的成長值，並於原本只有三個區間的 TCP Vegas 機制，修正成四個區間，以此方式提高吞吐量。首先 Quick Vegas 機制自訂了一個參數 *succ*，用來當 *cwnd* 成長的一個倍數乘積，並將原本 α 與 β 兩參數中間的區間修正為 α 至 $(\alpha+\beta)/2$ 為一區間，$(\alpha+\beta)/2$ 至 β 為另一區間。

當 *diff* < α 時，Quick Vegas 機制首先將 *succ* 參數加一，之後再判斷 *(β-diff)*succ* 是否大於目前的 *cwnd* 值，若比 *cwnd* 值還大，則將 *cwnd* 數值加一，反之則將 *cwnd* 加上 *(β-diff)/cwnd* 的大小；若 *diff* 介於 α 與 $(\alpha+\beta)/2$ 之間時，則將 *succ* 設為零，*cwnd* 數值加一；而當 *diff* 介於 $(\alpha+\beta)/2$ 與 β 之間時，則將目前 *cwnd* 減掉一，並將 *succ* 設為零；當 *diff* 大於 β 時，則將 *cwnd* 減掉 *(diff-(α+β)/2)* 的數值，同樣將 *succ* 設為零。

Quick Vegas 機制採用新的區間概念，並動態的依照目前的傳輸吞吐量差值 *diff* 作為增減 *cwnd* 的依據，的確有效的改善 TCP Vegas 機制 *cwnd* 成長不足的問題，但其 *cwnd* 值仍是過於保守，故應用在長距離高速傳輸網路之下仍是不足。

綜合以上各種兩種調整 TCP 機制的優缺點，本文決定採用量測 RTT 方式為調整壅塞視窗依據的 TCP Vegas 機制為基礎機制，並在分析上述修正 TCP Vegas 機制的優點後，得知要提高 TCP Vegas 機制在長距離高速傳輸網路的吞吐量，除了可提高 *cwnd* 的成長數值，調整 α 與 β 數值也可有效地提高 TCP Vegas 效能，故本人基於此兩種概念之下，提出本文主要的壅塞避免機制。

三、可支援長距離網路上高速傳輸之動態 TCP Vegas 設計

3.1 目前 TCP Vegas 在長距離高速傳輸網路之效能問題

以目前 TCP Vegas 來說，不適用於長距離高速傳輸網路環境上，主要原因大致如前文所敘述，會遇到兩項缺點：(1)*cwnd* 成長速度不夠完善，以致於無法有效利用長距離高速傳輸網路所有的頻寬。(2)當網路頻寬未達可用頻寬之上限且 RTT 差異值又座落在α、β之區間時，易造成 *cwnd* 不再增減而無法有效利用整個頻寬。故如何調整 *cwnd* 的成長速度與防止 *cwnd* 未達頻寬上限即陷入α、β區間是值得深入研究的問題。所以本文在前一章前先介紹較著名的 TCP 壅塞控制法及目前改進 TCP Vegas 的一些壅塞控制法，並同時在長距離高速傳輸網路的環境上，分別分析比較其網路效能。

圖 3-1 為測試環境的網路拓樸圖，其中 Source 與 router1 和 router2 與 Destination 之間的鏈路，頻寬為 1Gbps，延遲時間為 10ms，router1 與 router2 之間的鏈路，頻寬為 622Mbps，延遲時間為 110ms。

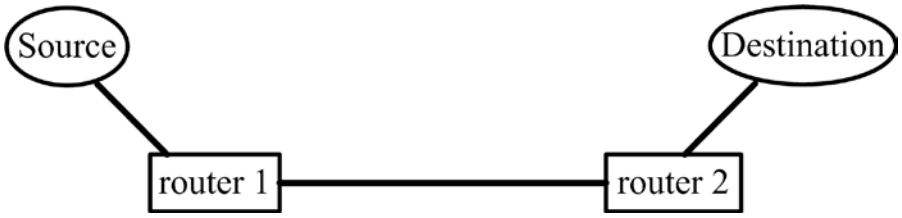

圖 3-1 網路拓樸圖

圖 3-2 可看出，Quick Vegas 機制與 Ada Vegas 機制由於修正過 *cwnd* 的成長速度，故可由圖中明顯看出比起 Vegas 與 Vegas-A 機制有著較高的 *cwnd* 數值，而 Vegas-A 機制所修正的調整α、β值部分應用在此網路情境則無法有效的發揮功用，主要是因其 *cwnd* 成長速度並未作修正的緣故。

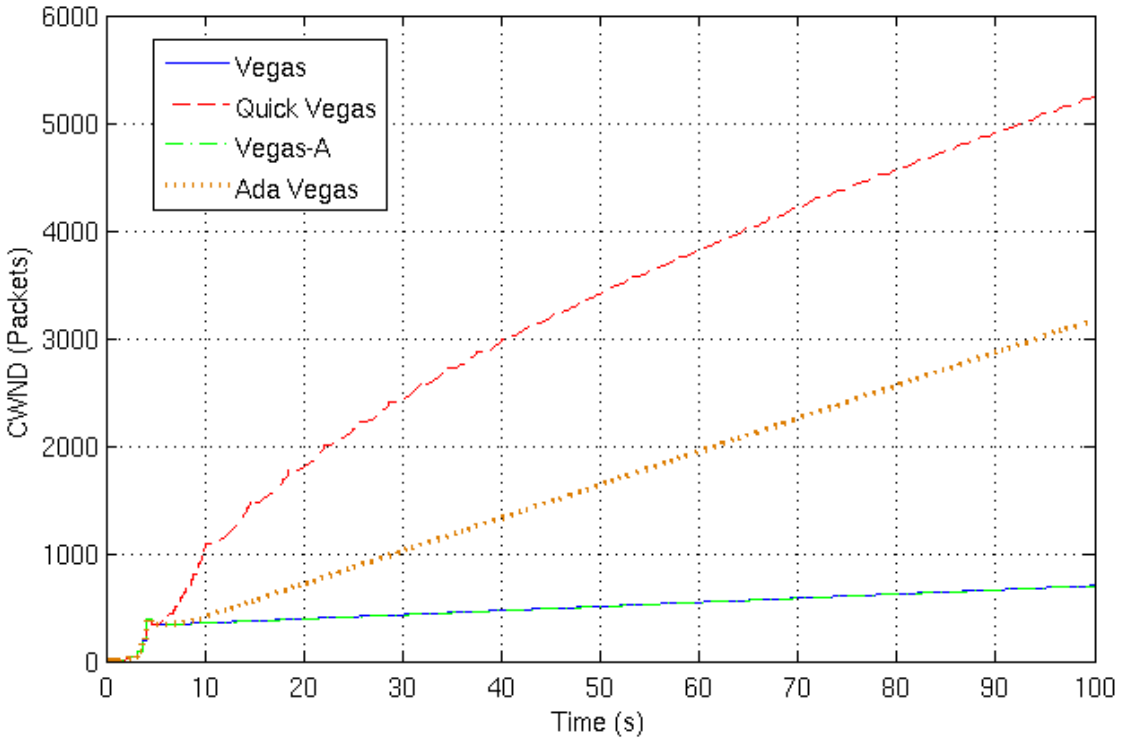

圖 3-2 不同 Vegas 版本比較圖

由表 3-1 得知，Quick Vegas 機制可由 622 Mbps 中達到 142.45 Mbps，而 Ada Vegas 機制則達到 86.84Mbps，而剩下兩種機制則只能達到 14.9 Mbps 左右，然而雖然 Quick Vegas 機制與 Ada Vegas 機制比起其他兩種機制有著更高的效能，但此四種機制皆無法有效的利用整個網路頻寬。

表 3-1 不同 Vegas 版本吞吐量表

機制名稱	吞吐量
Vegas	22.38 Mbps
Quick Vegas	142.45 Mbps
Vegas-A	22.35 Mbps
Ada Vegas	86.84 Mbps

由此測試結果可得知，要修正 TCP Vegas 機制在長距離高速傳輸網路上的吞吐量，基本上必須修正目前的 *cwnd* 成長速度，而如何修正 *cwnd* 成長速度則成為本章節主要探討的部分之一。

3.2 可支援長距離網路上高速傳輸之動態 TCP Vegas 設計方法

一般而言，單條連線的吞吐量如式(19)所示。其中 *packet_size* 為封包的大小，本文採用預設值 1KByte 大小，經由發送端傳送的 *cwnd* 數量與封包大小相乘再除以 *RTT* 時間，即可得到此一連線的吞吐量，乘上 *8* 則是將單位由 Byte 換算成 bit。因此，從此式子中可看出，不論是在長距離高速傳輸網路環境或是一般網路情境上，造成吞吐量很小的原因有兩點：(1) *RTT* 的數值很大，造成吞吐量很小；(2) *cwnd* 值很小，同樣也造成吞吐量的過小。而在長距離高速傳輸網路環境中，*RTT* 的值不但很大且目前 TCP Vegas 機制所能達到的 *cwnd* 數量亦很小，故此連線的吞吐量會非常的低。

$$Throughput = \frac{cwnd * packet\_size * 8}{RTT} \tag{19}$$

因此要增加連線吞吐量的方式則可由 *cwnd* 與 *RTT* 兩種網路參數中下手， 故本文採用 TCP Vegas 機制為基礎，根據量測到的 RTT 數值，比對實際與預期傳送吞吐量的差值 *diff*，依照座落於不同的 α、β 區間，修正其原本的 *cwnd* 成長方式，並從原本固定不變之 α、β 兩數值修正為動態調整的方式，以此方式修正不但可以改善 TCP Vegas 陷入固定 *cwnd* 大小的情境，也可提高固定 α、β 時的吞吐量大小，本文綜合以上兩種修正，以期望本文所提供之壅塞控制方式能在長距離高速傳輸網路下成為可靠且不錯的壅塞控制方法。

圖 3-3 為方法流程圖，期望能藉由修正 TCP Vegas 機制中，使發送端傳送 *cwnd* 數量提昇，來改善發送端傳送速率提升緩慢的狀況。

首先連線開始後，將測量每回合量測到的 *RTT*，並以預期傳送吞吐量與實際透過 RTT 量測到的吞吐量計算其差值 *diff*，從連線開始至此處皆與目前 TCP Vegas 機制無異，即是圖 3-3 中虛線框線之外的區間。

待計算出差值 *diff* 後，即進入圖 3-3 虛線框框內的部分，也就是本文主要修正的部分，依照 *diff* 值座落於不同 α、β 區間 的來估計目前網路頻寬是處於壅塞的程度為何，同時依不同的壅塞情況控制發送端傳送 *cwnd* 數量，之後再依壅塞情況調整 α、β 值，以此改善目前 TCP Vegas 機制在長距離高速傳輸網路上的效能直到結

束連線為止。

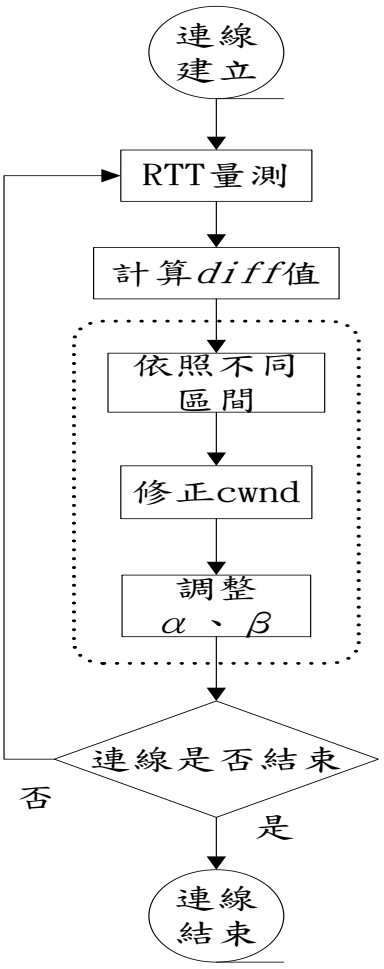

圖 3-3 修改後機制方法流程圖

目前 TCP Vegas 有著三個階段，分別是緩啟動、壅塞避免及快速重傳三個階段，其中本文主要修正的機制階段為壅塞避免階段，故另外兩種階段並不列入本文討論的範圍，而 TCP Vegas 壅塞避免階段機制的虛擬碼大致如下所示：

```
/*      CA   phase   */
IF    diff   <   α
      cwnd(k+1) = cwnd(k) + 1/cwnd(k);
else if      α  < diff < β
      cwnd(k+1) = cwnd(k);
else if      β  < diff
```

cwnd(k+1) = cwnd(k) – 1/cwnd(k);

End if

由上述的虛擬碼可看出，TCP Vegas 機制如何決定下一次發送端所需發送的 *cwnd* 值，主要是透過 *diff* 值與 α、β 比較後得知，以此方式在長距離高速傳輸網路上則會碰到以下兩點問題：

問題 1：

在高可用頻寬網路且 *RTT* 數值很大的情境之下，原本的 *cwnd* 成長數值需要經過 *cwnd* 個大小才會將 *cwnd* 增加一個，不難看出成長速度過慢而無法有效利用整個網路頻寬。

問題 2：

假設在可用頻寬尚未完全利用之前 *diff* 即進入 α、β 兩者之間的區間時，則會造成 *cwnd* 不會再作增減，此時將無法有效的利用整個網路頻寬。

由於 TCP Vegas 有著以上兩點問題存在，故本文將針對此兩點問題作一修正。

首先，本機制將原有 TCP Vegas 機制的三個區間，藉由新的區間分隔點($\alpha+\beta$)/2 分隔成四個區間，如圖 3-4 所示：

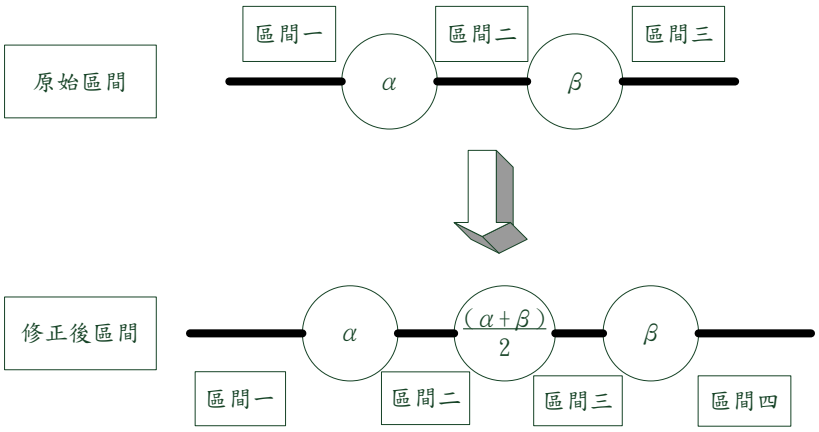

圖 3-4 新機制修正後區間圖

由圖 3-4 可看出，本文機制將原始 TCP Vegas 機制中 α、β 之間的區間，新增一分隔點($\alpha+\beta$)/2，將原本的區間二分割成修正後新的區間二與區間三，主要用意是期望能更進一步的區分出目前網路的壅塞情況，取代原本 TCP Vegas 機制直接

認定當預期傳送吞吐量與實際傳送吞吐量差值座落在α、β之間即為網路情況穩定，透過新分隔出來的區間，將可進一步對目前的網路情況作更有效率的調整。

本機制對此四區間的定義為：區間一為網路非常順暢區間，此時將快速的提升 *cwnd* 值；區間二為網路仍有可提升的空間但不如區間一順暢，故減緩提升 *cwnd* 區間，此時提升 *cwnd* 值將不如區間一為快；區間三為網路穩定區間，此時不針對 *cwnd* 的大小作調整，但仍會調整α、β以作網路的微調動作；區間四則代表網路即將進入壅塞區間，此時則會減少 *cwnd* 值以避免發生網路壅塞而造成封包遺失。本機制所有區間內詳細解說將在下段開始解說。

在區間一時，也就是當預期傳送吞吐量與實際傳送吞吐量差值 *diff* 小於α，此時會認定目前網路是處於非常順暢的情況，故當本機制判斷網路情況位在此區間一時將會大幅度的增加 *cwnd* 大小，以提高整個網路的吞吐量，之後再調整α、β的數值，期望能更有效地增加整個網路的使用效率，使得整個機制不會過早進入之後的區間而造成網路效能不足，而本機制位於區間一的虛擬碼如下所列：

```
/*    區間一    */
IF    diff   <  α
      P = P + 10;    // default P = 0
      cwnd(k) = cwnd(k) + (β-diff)*P;
      α  =  α  + 4;
      β  =  β  + 10;
End if
```

由上述區間一的虛擬碼可看出，本機制首先定義了變數 *P* 作為 *cwnd* 增加的乘積變數，主要用來提升 *cwnd* 的速度，而 *cwnd* 的成長值也由原本的 TCP Vegas 機制加 *1* 改為*(β- diff)*P* 作為下一次的成長值，以此方式不但可提升原本 *cwnd* 的速度，也由於動態的β、*diff* 數值，當 *diff* 值愈來愈大時，也就是網路預期傳送吞吐量與實際傳送吞吐量差異變大，此時 *cwnd* 成長的速度也會隨著β減 *diff* 的差值變小而減少，故此方式可更有效的因應網路情況的變化而不似之前採用固定的成長數值，無法因應整個網路情況。

修正完 *cwnd* 的成長值後，本機制將會調整 α、β 兩數值的大小，因為調整完後下一回合可能持續進入區間一，也有可能進入另外三個區間，而由虛擬碼可得知，α 數值成長並無 β 數值多，主要是希望此回合調整過後，進入區間二或區間三的機率能較其它兩區間高，假設下一回合真如期望的進入區間二甚至是區間三，如此一來便代表網路已慢慢接近頻寬上限或是穩定狀態，而進入到本機制緩慢提升 *cwnd* 或微調的區間。

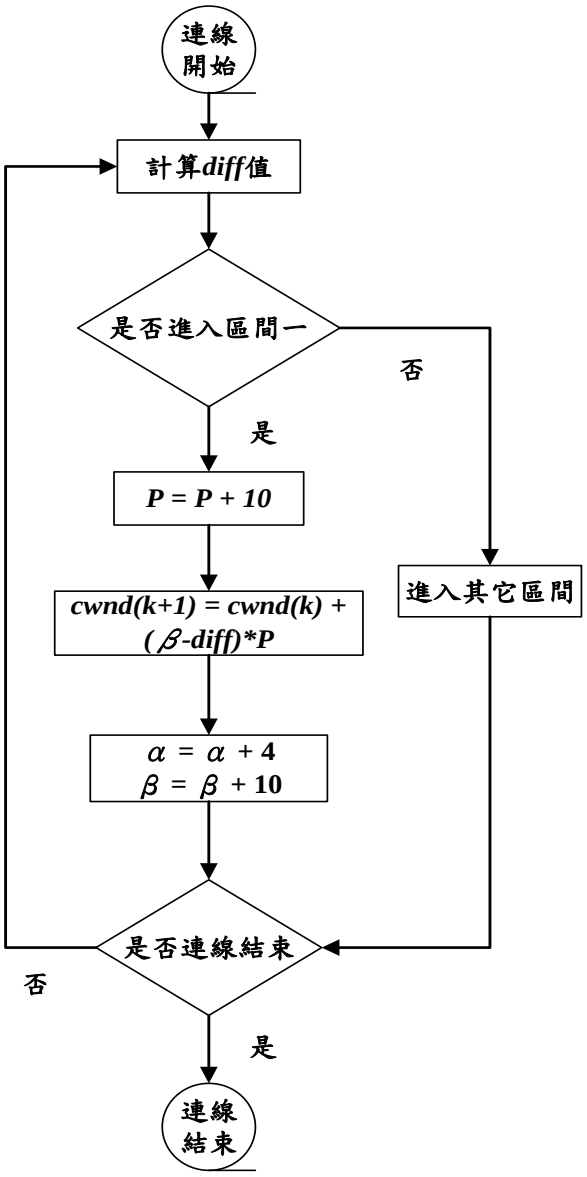

圖 3-5 區間一流程圖

本機制進入區間一的流程圖大致如圖 3-5 所示，其中計算預估與實際吞吐量差值 *diff* 之前預先計算實際 *RTT* 與最小 *RTT* 的 *BaseRTT* 部分因與原本 TCP Vegas 機制相同，故在此皆省略。本機制計算 *diff* 值後如判斷需進入區間一則開始區間一的機制，反之，則進入其它區間進行其它各區間不同的機制，當進入區間一後，首先會將 *P* 值加 *10*，之後增加*(β-diff)*P* 的 *cwnd* 成長數值，最後再將 α、β 分別加上 *4* 與 *10* 後進入下一回合或是因連線結束而結束連線。

若是進入區間二時，也就是當預期傳送吞吐量與實際傳送吞吐量差值 *diff* 介於 α 與$(\alpha+\beta)/2$ 之間，此時會認定目前網路仍可增加傳送吞吐量但不如區間一順暢，故需緩慢增加 *cwnd* 的情況，因此本機制判斷網路情況位在此區間二時將會減緩增加 *cwnd* 大小，以較緩慢的速度提高整個網路的吞吐量，之後再調整 β 的數值，同樣也期望能更有效地增加整個網路的使用效率，當整個機制因緩慢的成長 *cwnd* 時，期望有機會因網路變更順暢而回到區間一或是因網路可能造成壅塞而進到其它區間，而區間二的虛擬碼如下所列：

```
/*   區間二   */
IF   α  < diff < (α+β)/2
     P = 0;
     cwnd(k+1) = cwnd(k) + ( (α+β)/2 – diff );
     IF   β  > (α+10 )
          β  =  β  - 10;
     End if
End if
```

由上述區間二的虛擬碼可看出，本機制首先將區間一所定義的變數 *P* 設定為零，代表網路情況已不適用經由區間一成長過後的 *P* 值，雖然在區間二並未使用到 *P* 參數，但假如下一回合又回到區間一時，過高的 *P* 值將有可能造成在區間一的成長速度過快而造成網路壅塞，故在區間二先將 *P* 值設定為 *0* 避免此種情形發生。之後，將 *cwnd* 的成長值也由區間一增加*(β-diff)*P* 作為下一次的成長值修正為 *(α+β)/2 – diff)*，以此方式不但可較區間一緩慢的提升 *cwnd* 的速度，也由於動態的 α、β 及 *diff* 數值，當 *diff* 值愈來愈大時，也就是網路預期傳送吞吐量與實際傳送吞吐量差異變大，此時 *cwnd* 成長的速度也會隨著*(α+β)/2* 減掉 *diff* 的差值變

小而減少，故此方式也可更有效的因應網路情況的變化。

如同區間一，區間二在調整完 *cwnd* 後，也會對 α 、 β 作一調整，不同的是在區間二的調整方面，只針對 β 作調整，調整方式則將目前的 β 減掉 *10* 為下一回合的 β 值，主要用意是希望由之前區間所成長之後的 β 在此區間內可慢慢收斂，避免過高的 α 、 β 區間造成路由器間佇列的封包堆積太多，此外收斂後的區間大小期望更能反應目前網路情況。

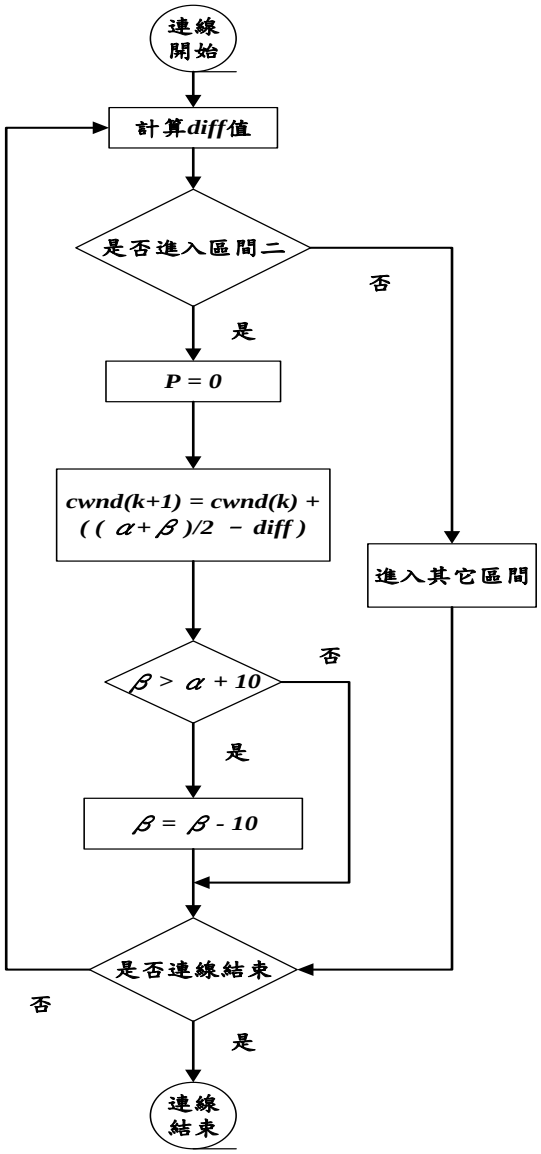

圖 3-6 區間二流程圖

圖 3-6 所示為本機制進入區間二的流程圖，其中計算預估與實際吞吐量差值 *diff* 之前預先計算實際 *RTT* 與最小 *RTT* 的 *BaseRTT* 部分因與原本 TCP Vegas 機制相同，故在此亦省略。本機制計算 *diff* 值後如判斷需進入區間二則開始區間二的機制，反之則進入其它區間進行其它各區間不同的機制，當進入區間二後，首先會將 *P* 值設定為 *0*，之後增加 *(α+β)/2 – diff* 的 *cwnd* 成長數值，最後再將判斷 *β* 是否有大於 *α* 加 *10*，若有則將 *β* 扣掉 *10* 後進入下一回合或是因連線結束而結束連線。

當進入區間三後，也就是當預期傳送吞吐量與實際傳送吞吐量差值 *diff* 介於 *(α+β)/2* 與 *β* 之間，此時會認定目前網路是處於穩定的情況，故當本機制判斷網路情況位在此區間三時將會不增加也不減少 *cwnd* 的大小，主要是本機制認定進入此區間則表示網路已快進入壅塞的情況，故判定再增加 *cwnd* 會造成網路產生壅塞的情況，故在此區間並不作增加 *cwnd* 的動作，雖然此區間不增減 *cwnd* 的大小，但仍會修正 *α* 及 *β* 的數值，主要用意是希望藉由修正 *α* 及 *β* 兩數值後，能使網路更趨向於穩定的狀態，而整個區間三的虛擬碼如下所列：

```
/*   區間三   */
IF   (α+β)/2 < diff <  β
     P = 0;
     IF   α  > 4
          α  =  α- 4;
     End if
     IF   β  > (α+2 )
          β  =  β  - 2;
     End if
End if
```

由上述區間三的虛擬碼可得知，首先同樣將 *P* 值設定為 *0*，用意與區間二時一樣，避免下一回合又回到區間一時成長速度過快，而造成網路進入壅塞，而在此區間內並不作 *cwnd* 的調整，直接進入 *α* 與 *β* 的調整。在此區間三，*α* 會先判斷是否大於 *4*，若大於 *4* 則 *α* 會直接扣掉 *4*，若小於 *4* 則 *α* 不作增減；之後再判斷 *β* 是否比 *α* 還多 *2*，若是 *β* 則會扣掉 *2*，反之則同樣 *β* 也不作增減。主要用意是在於收斂整個 *α*、*β* 值，也就等同於收斂全部區間的範圍，其中 *α*、*β* 收斂的大小不一，

主要是讓收斂之後進入區間二與區間三的機率比其它區間大，因此在此區間作一微調的動作。

圖 3-7 所示為本機制進入區間三的流程圖，其中計算預估與實際吞吐量差值 *diff* 之前預先計算實際 *RTT* 與最小 *RTT* 的 *BaseRTT* 部分因與原本 TCP Vegas 機制相同，故在此亦省略。本機制計算 *diff* 值後，如判斷需進入區間三則開始區間三的機制，反之則進入其它區間進行其它各區間不同的機制，當進入區間三後，首先會將 *P* 值設定為零，之後並未作任何 *cwnd* 成長數值的變動，接著判斷 α 是否比 *4* 還大，若大於 *4* 則 α 扣掉 *4*，若無則直接判斷 β 是否有大於 α 加 *2*，若有則將 β 扣掉 *2* 後進入下一回合或是因連線結束而結束連線。

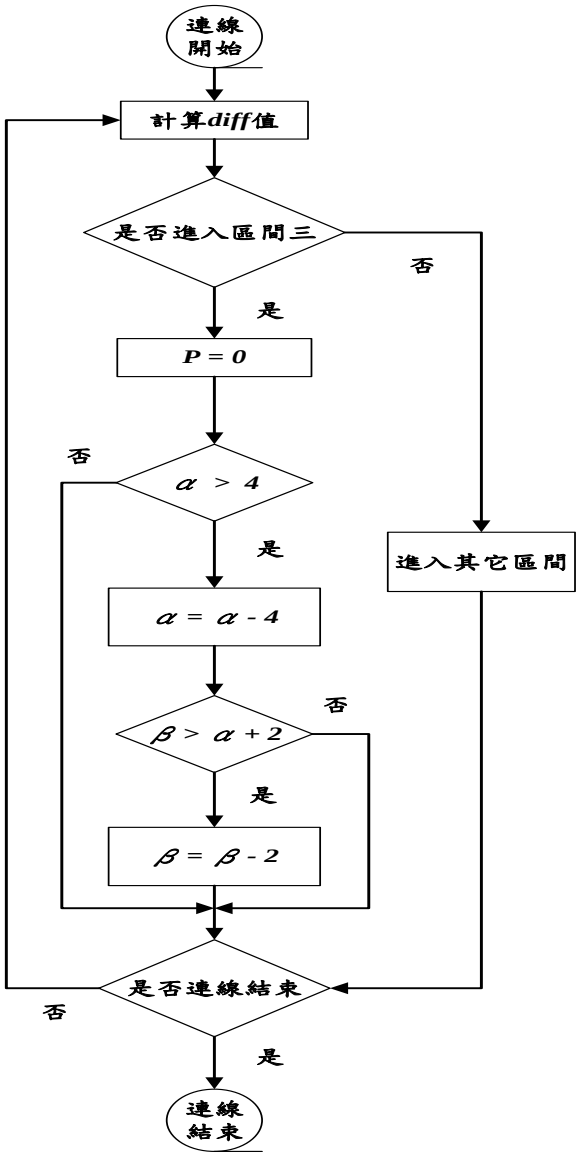

圖 3-7 區間三流程圖

最後當進入區間四後，也就是當預期傳送吞吐量與實際傳送吞吐量差值 *diff* 大於 β 之後，此時會認定目前網路再增快速度將會是處於壅塞的情況，故當本機制判斷網路情況位在此區間四時將會開始減少 *cwnd* 的大小，主要是本機制認定進入此區間則表示網路已經接近進入壅塞的臨界值的情況，故判定再增加 *cwnd* 一定會造成網路產生壅塞的情況，故在此區間開始執行減少 *cwnd* 的動作，之後並不作 α 或 β 值的調整，主要是視為目前的 α 、 β 為最高的臨界值，保持目前的數值即可，待減少 *cwnd* 後，期望由其它區間作更佳的 α 、 β 值調整，而整個區間四的虛

擬碼如下所列：

```
/*   區間四   */
IF   β < diff
     P = 0;
     IF (diff-β) > cwnd(k)
          cwnd(k+1) = 2;
     else
          cwnd(k+1) = cwnd(k) – (diff – β);
     End if
End if
```

由上述區間四的虛擬碼可得知，首先同樣也將 *P* 值設定為 *0*，用意與區間二及區間三時一樣，避免假設下一回合回到區間一時成長速度過快，而造成網路進入壅塞，由於進入區間四會判斷網路已進入壅塞的臨界值，故在此區間內會先判斷 *diff-*β 的數值是否比目前的 *cwnd* 值還大，若是則會將下一回合的 *cwnd* 值設定為二，反之則直接扣掉 *diff-*β，而在此區間內並不調整α與β值，主要是認定目前的α與β值是接近壅塞的臨界值而不作修正。

圖 3-8 所示為本機制進入區間四的流程圖，其中計算預估與實際吞吐量差值 *diff* 之前預先計算實際 *RTT* 與最小 *RTT* 的 *BaseRTT* 部分因與原本 TCP Vegas 機制相同，故在此亦省略。本機制計算 *diff* 值後如判斷需進入區間四則開始區間四的機制，反之則進入其它區間進行其它各區間不同的機制，當進入區間四後，首先會將 *P* 值設定為 *0*，之後判斷 *diff-*β是否比目前的 *cwnd* 還大，若大於 *cwnd* 則將 *cwnd* 扣掉 *diff-*β的大小，若無則直接設定 *cwnd* 值為 *2*，而α與β並不作更動就進入下一回合或是因連線結束而結束連線。

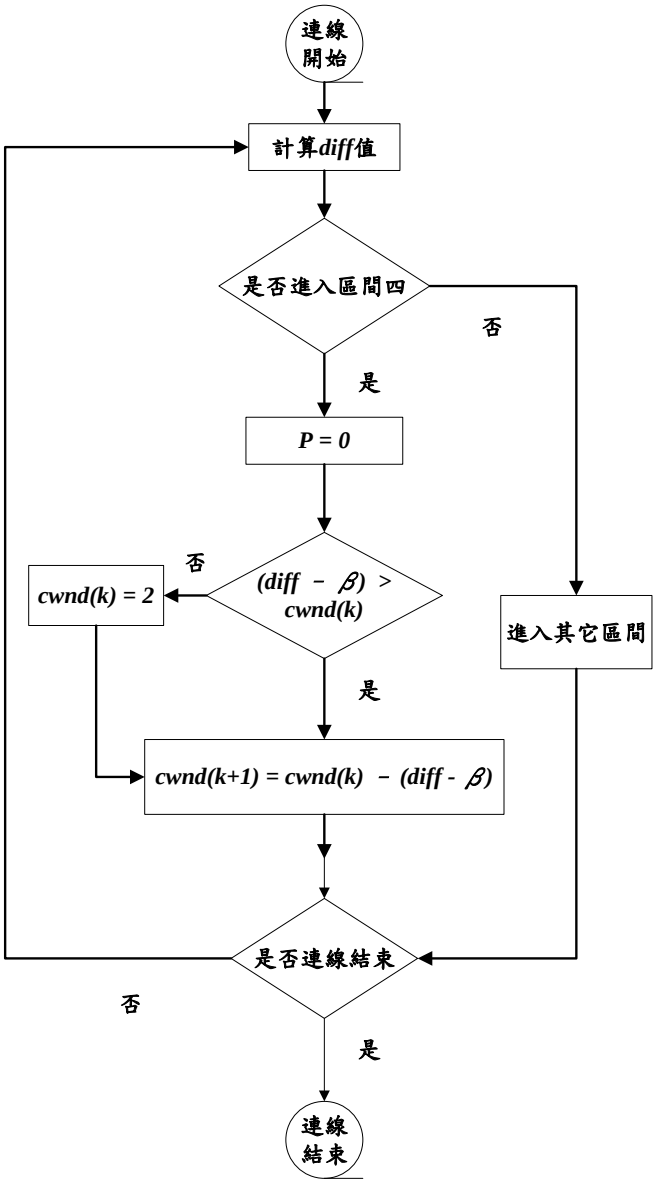

圖 3-8 區間四流程圖

而當預期傳送吞吐量與實際傳送吞吐量差值 *diff* 等同於分隔點時，也就是 α、β 以及 $(\alpha+\beta)/2$ 時，在本機制內設定為期望 α、β 收斂後能進入的數值，也就是說當 *diff* 值與分隔點相同時，本機制並不對 *cwnd*、α 及 β 三個參數作任何的變更，唯有本機制自行設定的參數 *P* 值將設定為 *0*，其用意也與區間二至區間四相同。

整合上述四個區間與分隔點的機制後，即為本文所提之完整壅塞避免機制，整個機制虛擬碼如下所示：

```
/*          CA phase        */
IF    diff   <  α            // 區間一
      P = P + 10;    // default P = 0
      cwnd(k+1) = cwnd(k) + (β-diff)*P;
      α  =  α  + 4;
      β  =  β  + 10;
else if      α  < diff < (α + β)/2   // 區間二
      P = 0;
      cwnd(k+1) = cwnd(k) + ( (α + β)/2 – diff );
      IF    β  > (α +10 )
            β  =  β  - 10;
      End if
else if      (α + β)/2 < diff < β          // 區間三
      P = 0;
      IF    α  > 4
            α  =  α - 4;
      End if
      IF    β  > (α +2 )
            β  =  β  - 2;
      End if
else if      β < diff         // 區間四
      P = 0;
      IF (diff- β) > cwnd(k)
            cwnd(k+1) = 2;
      else
            cwnd(k+1) = cwnd(k) – (diff – β);
      End if
else                           // 分隔點
      P = 0;
End if
```

從上頁完整壅塞避免機制的虛擬碼可看出，計算出差值 *diff* 後，依照其大小與 α 及 β 作比較後，進入不同的區間，以此方式進行每個回合調整 *cwnd* 的依據，而完整的機制流程圖則如圖 3-9 所示。

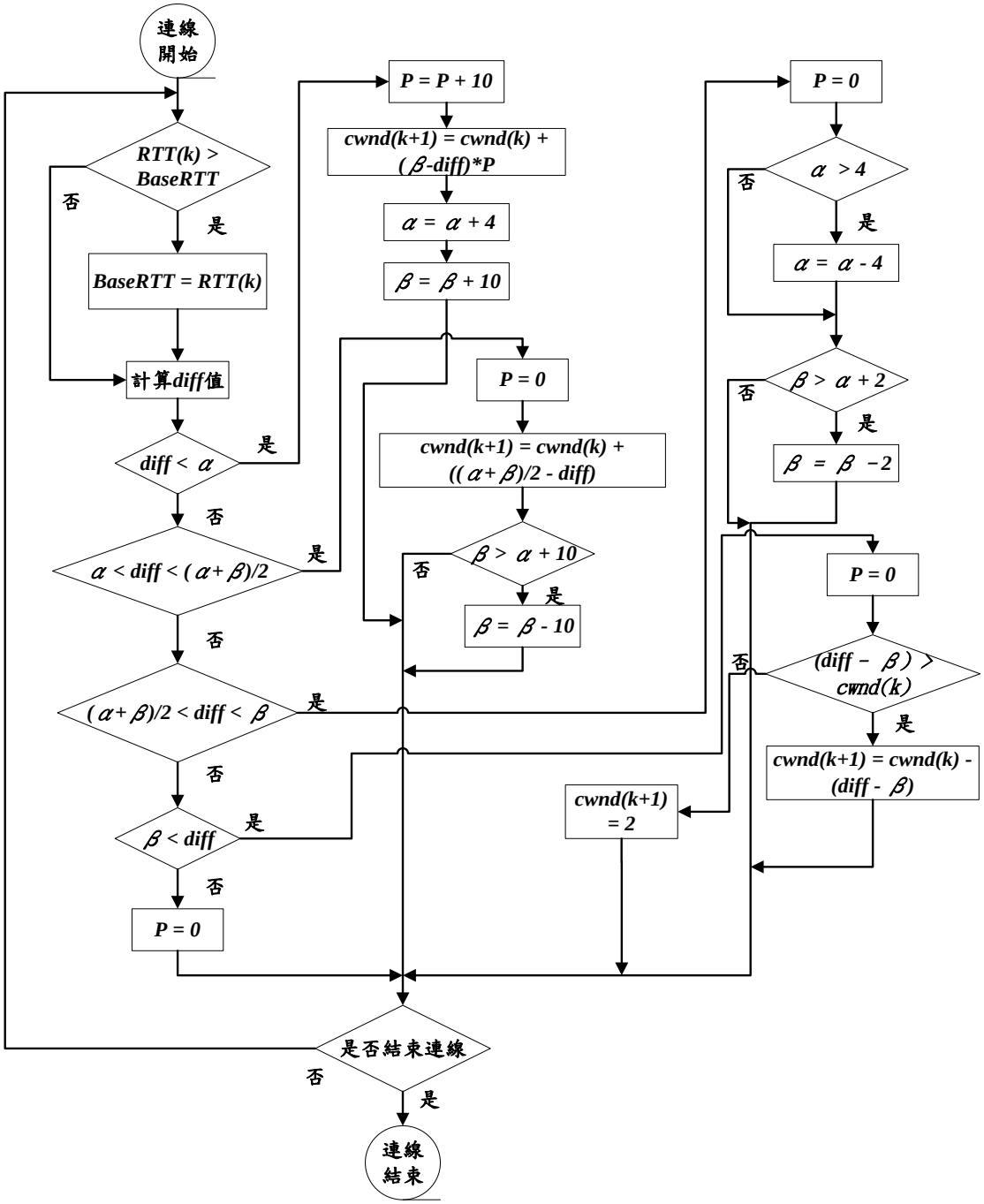

圖 3-9 完整機制流程圖

四、模擬與分析

4.1 實驗目標與環境

為驗證本文上一節所提出的機制是否可行，在此章節將利用 NS2[18]模擬工具設計模擬情境，依照下列七項目標，作深入探討：

1. 長距離高速傳輸網路環境，本文所提出擁擠避免機制是否能有效改善目前 TCP Vegas 效能。
2. 長距離高速傳輸網路環境，本文所提出擁擠避免機制是否能有效比目前改進 TCP Vegas 效能的方式更佳。
3. 依照不同的傳輸距離，本文所提出壅塞避免機制是否能確實達到網路瓶頸鏈路的頻寬吞吐量。
4. 依照不同的頻寬大小，本文所提出壅塞避免機制是否能確實達到各網路瓶頸鏈路的頻寬吞吐量。
5. 背景流量對本文所提出壅塞避免機制之影響。
6. 多條連線之下，本文所提出壅塞避免機制的影響分析。

為了驗證以上幾項目標，在此設計了一個網路拓樸，發送端數目依照各測試環境不同而不同，接收端也如同發送端一樣，線路連接由各發送端編號經由兩個路由器連接到各相同編號的接收端，而發送到及接收端到路由端之間的傳遞延遲皆為 10ms，網路頻寬為 1Gbps，而路由端之間的傳遞延遲為 Y ms，Y 值由不同的測試情境作不同的延遲時間設定，而網路頻寬則為 X Mbps，X 值也同樣由不同的測試情境作不同頻寬設定，而路由之間的緩衝區大小以不造成封包遺失為設定依據。整個網路拓樸如圖 4-1 所示。

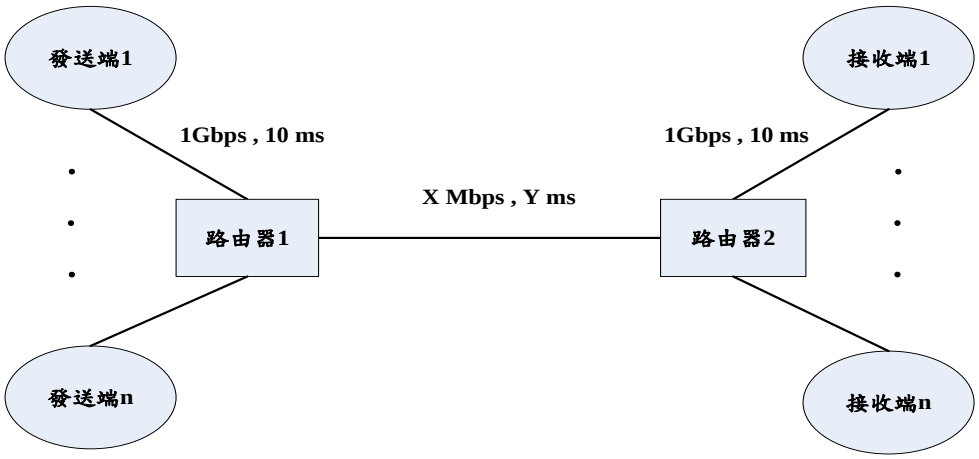

圖 4-1 網路拓樸圖

4.2 實驗結果與分析

I. 為了驗證及分析目標一，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處採用點對點的單一連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，X 值設定為 622，Y 值設定為 110，模擬時間設定為 100 秒。

步驟 2：分別以 TCP Vegas 機制與本文所提出之新機制執行此模擬環境。

步驟 3：依模擬情形分別記錄其 *cwnd* 成長情況與 RTT 之情況。

步驟 4：計算其平均吞吐量，並作分析比較。

圖 4-2 為目標一經由 NS2 模擬之後比較本文機制與 TCP Vegas 的 *cwnd* 成長情況比較，由圖中可得知，本文機制在 54.9 秒時，即達到頻寬上限 622Mbps 而不再增加 *cwnd*，反觀 TCP Vegas 機制則處於龜速成長的狀態，至模擬時間終止 100 秒時，*cwnd* 只能增加至 701 的大小，對於整個網路頻寬的使用率非常的低。圖 4-3 則為本機制與 TCP Vegas 機制模擬時間內 RTT 的變化圖，由圖中可看出，大約在 4 秒多時本機制與 TCP Vegas 機制皆由緩啟動階段進入壅塞避免階段，故 RTT 稍微提升，之後 TCP Vegas 則因為吞吐量提升緩慢的緣故，至模擬時間結束都因網路順暢而未提高 RTT 數值，一直保持在 260ms，反觀本機制，因提升 *cwnd* 的成長數值，且又動態的調整 α 、 β 數值，故停留在 Queue 裡的封包數目稍微增加，因而

增加了 RTT 的時間，而模擬時間在 54 秒多時，RTT 時間突然增高了 10ms 左右，之後即幾乎停於 270ms，造成此現象的主要是由於網路實際頻寬已接近上限，而預期吞吐量與實際吞吐量的差值 *diff* 仍在本機制區間一內，故造成此現象，而當達到頻寬上限後，*diff* 值上升至先後進入區間二與區間三，因在區間二時 *cwnd* 成長速度較慢而區間三則不增加傳送 *cwnd* 的大小，故看起來 *cwnd* 均保持不變，只剩下 α、β 兩變數作微調，在之後的時間內雖然看起來 RTT 均未變化，但實際上正慢慢減少當中。

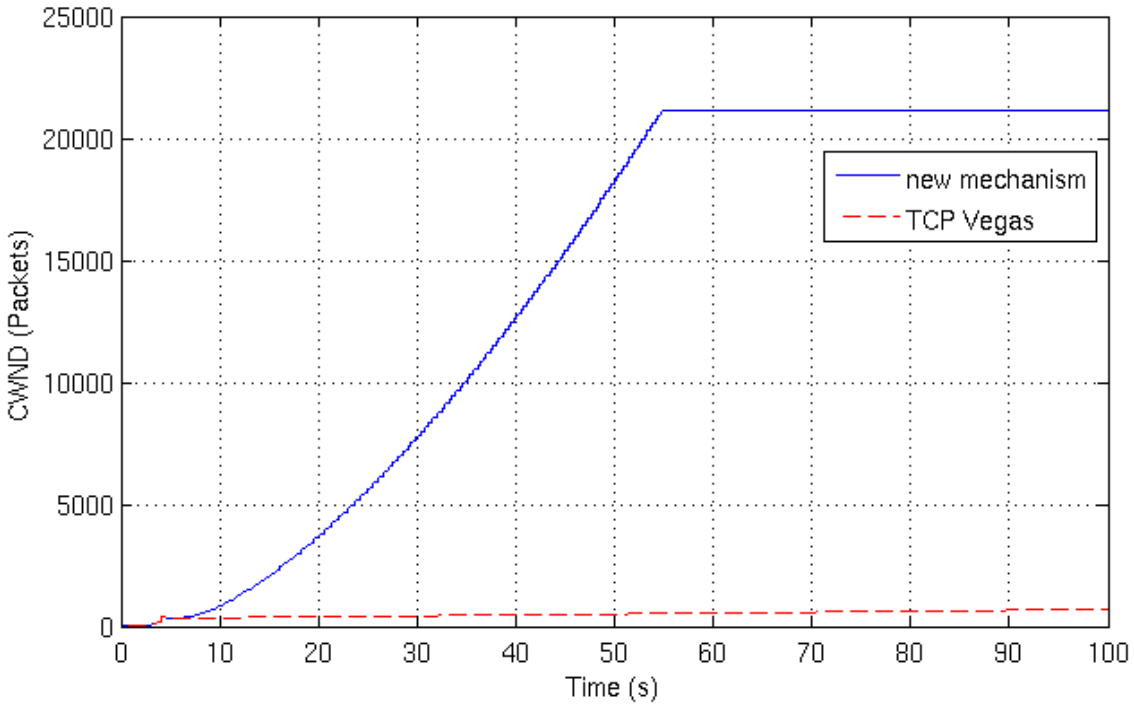

圖 4-2 本文機制與 TCP Vegas 之 cwnd 比較圖

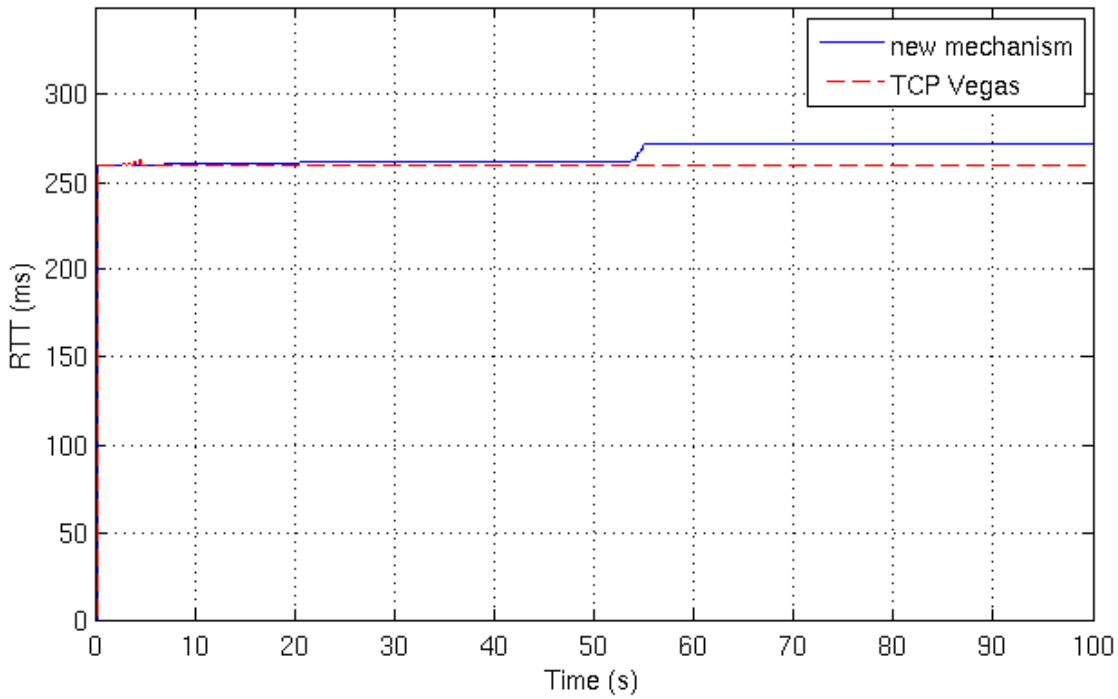

圖 4-3 本文機制與 TCP Vegas 之 RTT 比較圖

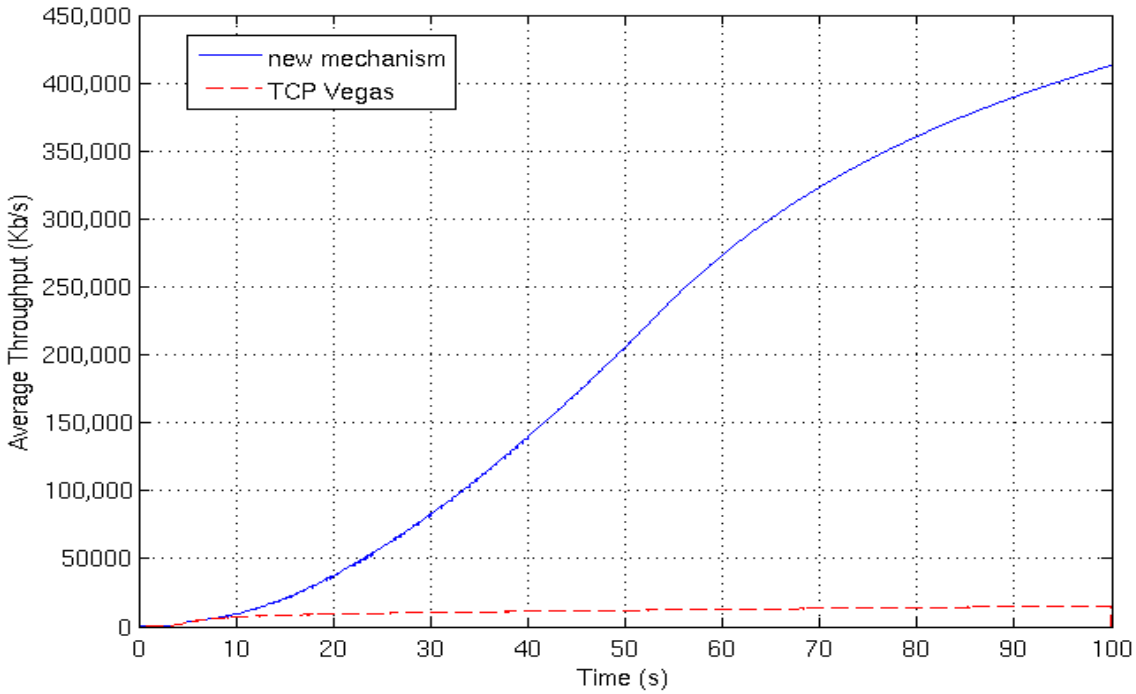

圖 4-4 本文機制與 TCP Vegas 之平均吞吐量比較圖

圖 4-4 為本文機制與 TCP Vegas 吞吐量比較，計算方式如式(20)，由當回合的 *cwnd* 數值乘上封包大小之後再除以到當回合為止的時間，即式 20 中的 *time* 參數，由圖 4-4 可明顯看出，大約在十秒內本機制與 TCP Vegas 機制的吞吐量均差不多，主要是因為在緩啟動階段本機制與 TCP Vegas 機制一樣的緣故，之後因本機制快速提升 *cwnd* 數值以因應整個網路頻寬，故成長速度遠比 TCP Vegas 機制快速，過了 54 秒後，由於達到頻寬上限 622Mbps，所以成長速度有趨於緩慢的現象。

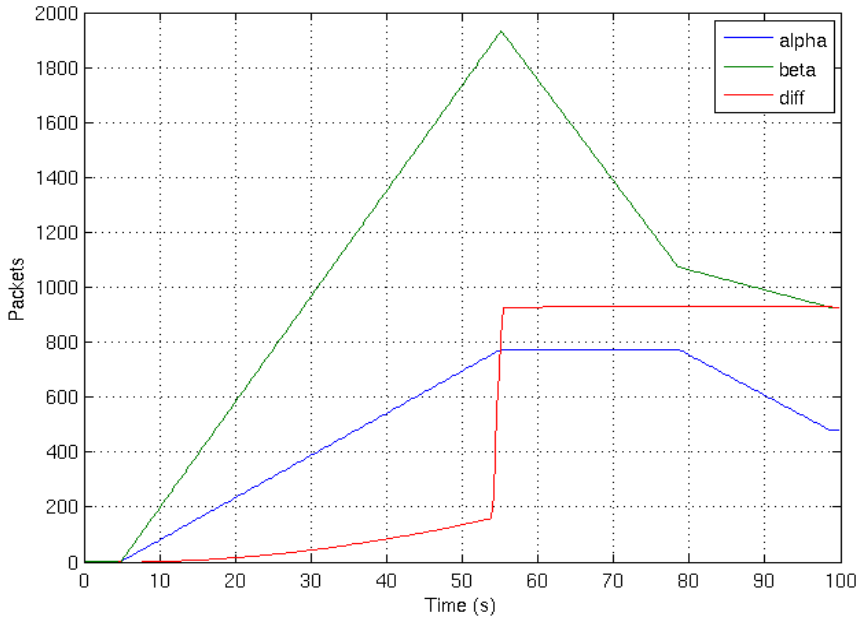

圖 4-5 本文機制之 α 、 β 及 *diff* 成長圖

由圖 4-5 可以看出，本文機制在瓶頸頻寬 622Mbps 時 α 、 β 及 *diff* 成長的情況，在 55 秒之前皆處於第一區間，故 α 、 β 呈現快速成長狀態，此時 α 、 β 的區間不斷的拉開，待 55 秒後至 78 秒間處於第二區間，此時 α 處於不變的情況，而 β 不斷的下降修正，待過了 78 秒則進入第三區間， α 、 β 不斷的向下修正，且修正的期間逐漸拉開 α 、 β 的區間大小，待 98 秒之後則在第四區間與第三區間不斷的來回，由於本文模擬時間為 100 秒，故之後調整 α 、 β 的狀態並未呈現。

$$Throughput = \frac{cwnd * packet\_size}{time} \tag{20}$$

表 4-1 則為本機制與 TCP Vegas 機制在模擬時間 100 秒內經由式(20)所算得之吞吐量，由表中可明顯看出本機制達到 412.32Mbps 的吞吐量大小，而 TCP Vegas 機制只能達到 15.33Mbps，在網路使用率上本機制為 TCP Vegas 機制的 26.9 倍，足可驗證本機制比 TCP Vegas 機制在長距離高速傳輸網路上網路頻寬的使用上更有效率。

表 4-1 本文機制與 TCP Vegas 平均吞吐量比較表

機制名稱	平均吞吐量(100 秒內)
TCP Vegas	15.33 Mbps
New mechanism	412.32 Mbps

II. 為了驗證及分析目標二，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處同樣採用點對點的單一連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，X 值設定為 622，Y 值設定為 110，模擬時間設定為 100 秒。

步驟 2：分別以本文所提出之新機制與 Quick Vegas 機制、Vegas-A 機制以及 Ada Vegas 機制執行此模擬環境。

步驟 3：依模擬情形分別記錄其 *cwnd* 成長情況與 RTT 之情況。

步驟 4：計算其平均吞吐量，並作分析比較。

圖 4-6 為本文機制與其它三種機制 *cwnd* 的比較圖，由圖中可看出，Quick Vegas 機制由於修正過 *cwnd* 的成長速率，故在長距離高速傳輸網路環境之下 *cwnd* 成長速度比 TCP Vegas 機制還快，而 Ada Vegas 機制由於較 Quick Vegas 機制成長 *cwnd* 保守，故比 Quick Vegas 機制所達到的 *cwnd* 值還低，但反觀 Vegas-A 機制，由於此機制並未修正 *cwnd* 的成長速度，而是單純修正 α、β 的數值，故在長距離高速傳輸網路之下成長速度與 TCP Vegas 機制並無太大差異，也造就其使用網路的效率不佳，且不論是 Quick Vegas 機制、Ada Vegas 機制或是 Vegas-A 機制，此三種機制均未能達到此網路拓樸的上限頻寬 622Mbps，不像本文機制可確實達到此網路拓樸的頻寬上限。

圖 4-7 則為本文機制與其它三種機制的 RTT 比較圖，由圖中可看出由於其它三種機制皆未達到頻寬上限，成長的速度不足以讓 Queue 產生等待的時間，故在 RTT 的數值上並未有太大的變化，除了在緩啟動結束時讓 RTT 稍微增高，其它時間兩機制均維持在 260ms。

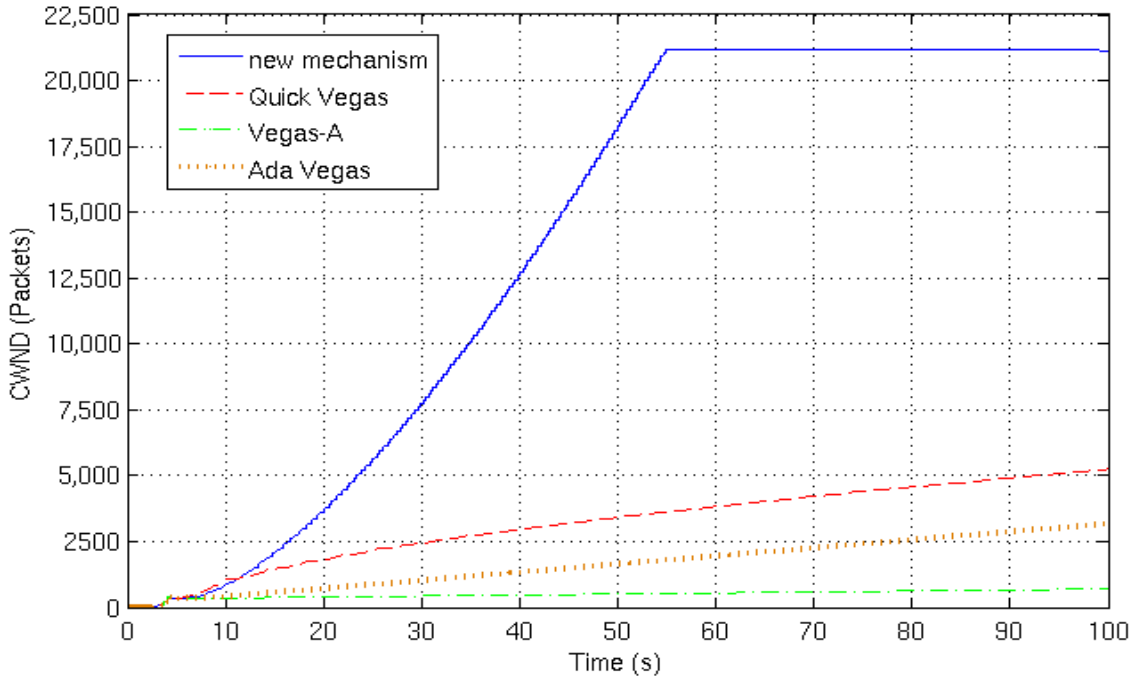

圖 4-6 本文機制與其它三種機制之 cwnd 比較圖

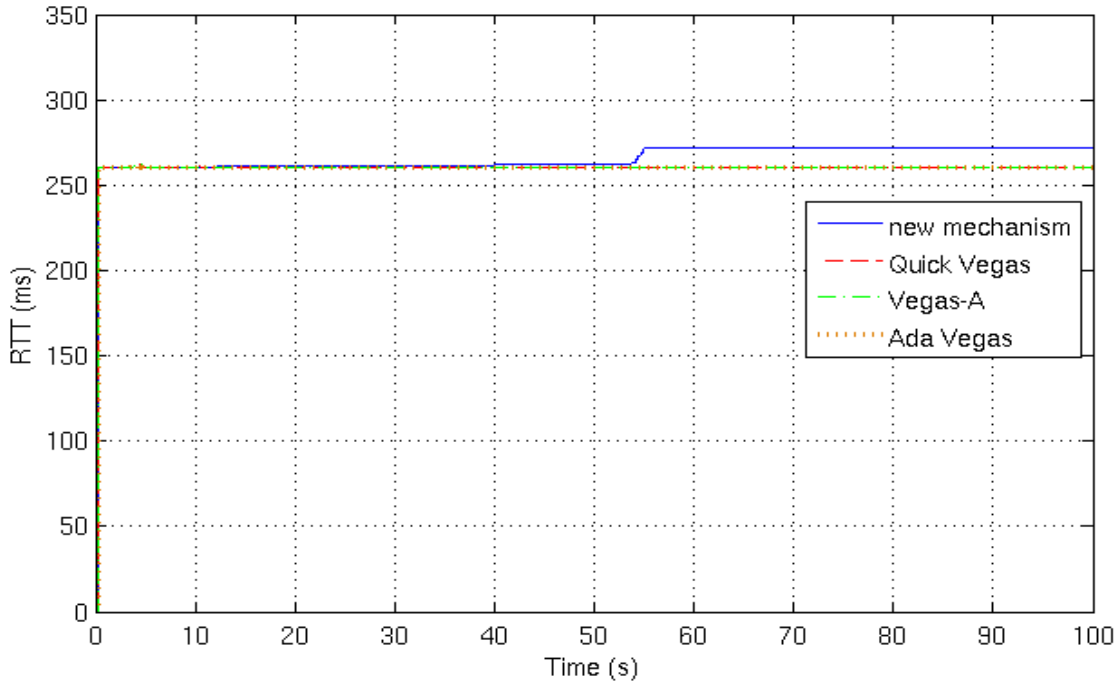

圖 4-7 本文機制與其它三種機制之 RTT 比較圖

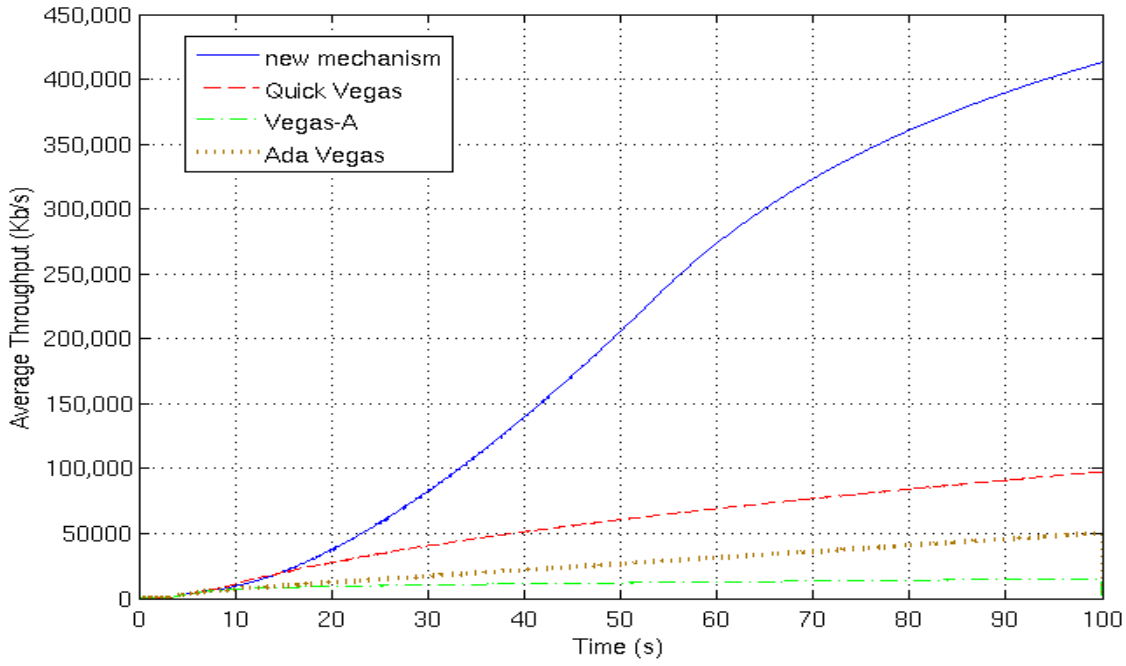

圖 4-8 本文機制與其它三種機制之平均吞吐量比較圖

圖 4-8 為本文機制與其它三種機制吞吐量比較，計算方式如同式(20)，由圖 4-8 可明顯看出，大約在十秒內本機制與其它三種機制的吞吐量均差不多，主要是因為在緩啟動階段本機制與其它三種機制皆與 TCP Vegas 一樣的緣故，之後因本機制快速提升 *cwnd* 數值以因應整個網路頻寬，故 *cwnd* 成長速度遠比其它兩種機制快速，Quick Vegas 機制雖然也提高了 *cwnd* 的成長，但由於其過於保守的成長速度，導致在 15 秒左右吞吐量便與本文機制拉開差距，之後則呈現固定的成長速度持續成長，Ada Vegas 機制則以更保守的速度成長，故與 Vegas-A 機制一樣，大約在 8 秒左右即拉開整個差距，最後 Vegas-A 機制則如同 TCP Vegas 般，採用龜速的方式增加 *cwnd*。

表 4-2 為本文機制與其它三種機制的吞吐量比較表，計算方式如同表 4-1，由表 4-2 中可知 Quick Vegas 機制因提高 *cwnd* 成長速度，故平均吞吐量比起 TCP Vegas 機制有明顯改善，100 秒內可達到 97.35Mbps，但仍無法達到頻寬上限 622Mbps，反觀 Vegas-A 機制，因未修正 *cwnd* 的成長速度，故平均吞吐量與 TCP Vegas 機制並無太大差別，但不論是 Quick Vegas 機制或是 Vegas-A 機制，本文所提之機制均有效的改善了目前改進 TCP Vegas 機制的效能，分別改進了 4.24 倍、26.94 倍與 8.22 倍。由此可驗證本文所提之機制確實比改善目前修正 TCP Vegas 效能的其它機制更佳。

表 4-2 本文機制與其它三種機制平均與瞬時吞吐量比較表

機制名稱	**平均吞吐量(100 秒內)**	**瞬時吞吐量(100 秒時)**
New mechanism	412.32 Mbps	622 Mbps
Quick Vegas	97.35 Mbps	142.45 Mbps
Vegas-A	15.30 Mbps	22.35 Mbps
Ada Vegas	50.18 Mbps	86.84 Mbps

III. 為了驗證及分析目標三，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處同樣採用點對點的單一連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，X 值設定為 155，Y 值分別設定為 100、200、300 及 400，模擬時間設定增加為 200 秒。

步驟 2：以本文所提出之新機制分別執行上述不同 Y 值的模擬環境。

步驟 3：依模擬情形分別記錄其 *cwnd* 成長情況與 RTT 之情況。

步驟 4：計算其平均吞吐量，並作分析比較。

圖 4-9 為本文機制在不同距離即不同 RTT 之 *cwnd* 比較圖，由圖中可明顯看出，當 RTT 數值越高，本文機制達到瓶頸鏈路之頻寬的時間也就越久，RTT 從 240ms 時只需大約 21 秒即可達到瓶頸鏈路的頻寬至 RTT 為 840ms 則需花費 158 秒左右才可達到瓶頸鏈路的頻寬。其中可看出 RTT 在 240ms 與 440ms 時，時間越久後期的 *cwnd* 有逐步下降的趨勢，主要是由於進入本文機制第三區間中調整 α、β 數值的緣故，可由圖 4-10 中比對得知，本文機制達到瓶頸鏈路的頻寬後 RTT 會增加幾十 ms 左右，之後再如同 240ms 與 440ms 的情況一樣，慢慢作減少 RTT 數值的調整，最後如同 240ms 一樣，將 RTT 修正為最低值 240ms，而且在調整 RTT 大小的整個過程中，吞吐量的大小並不會受到調整 RTT 的影響而減少。

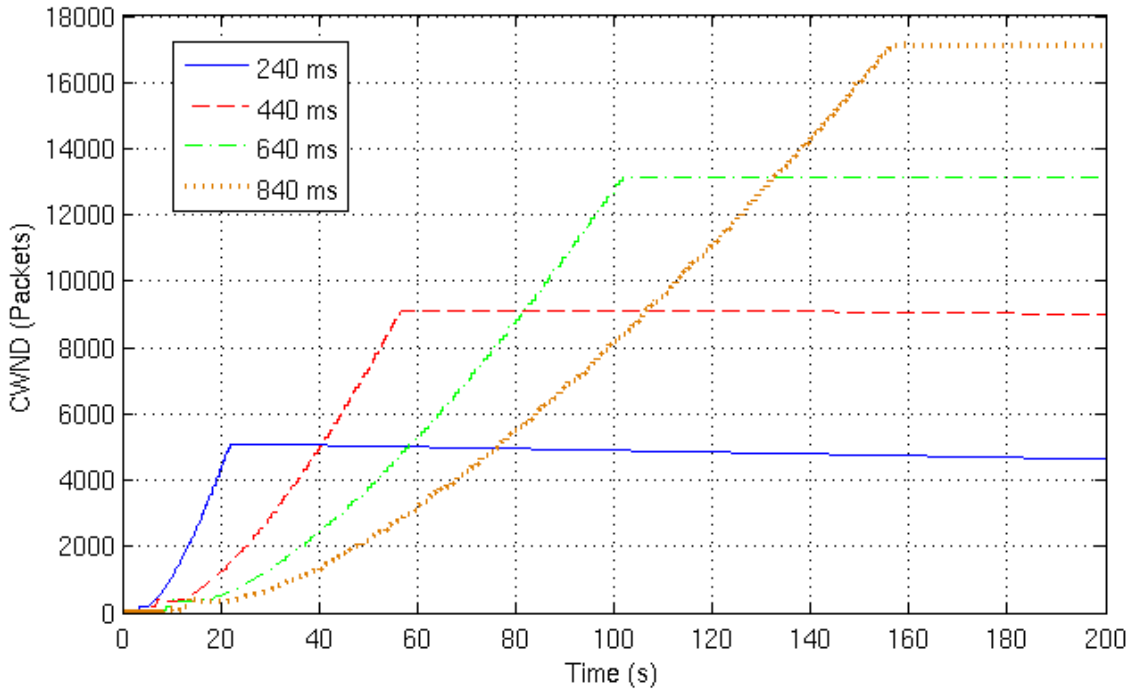

圖 4-9 本文機制在不同 RTT 之 cwnd 比較圖

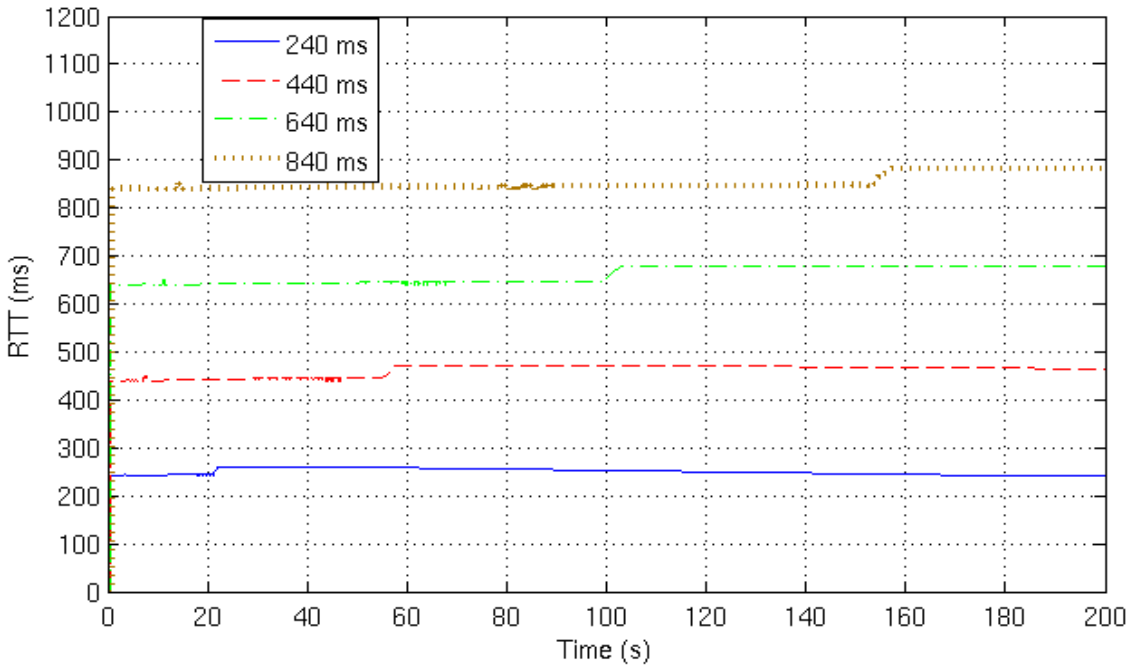

圖 4-10 本文機制在不同 RTT 之 RTT 比較圖

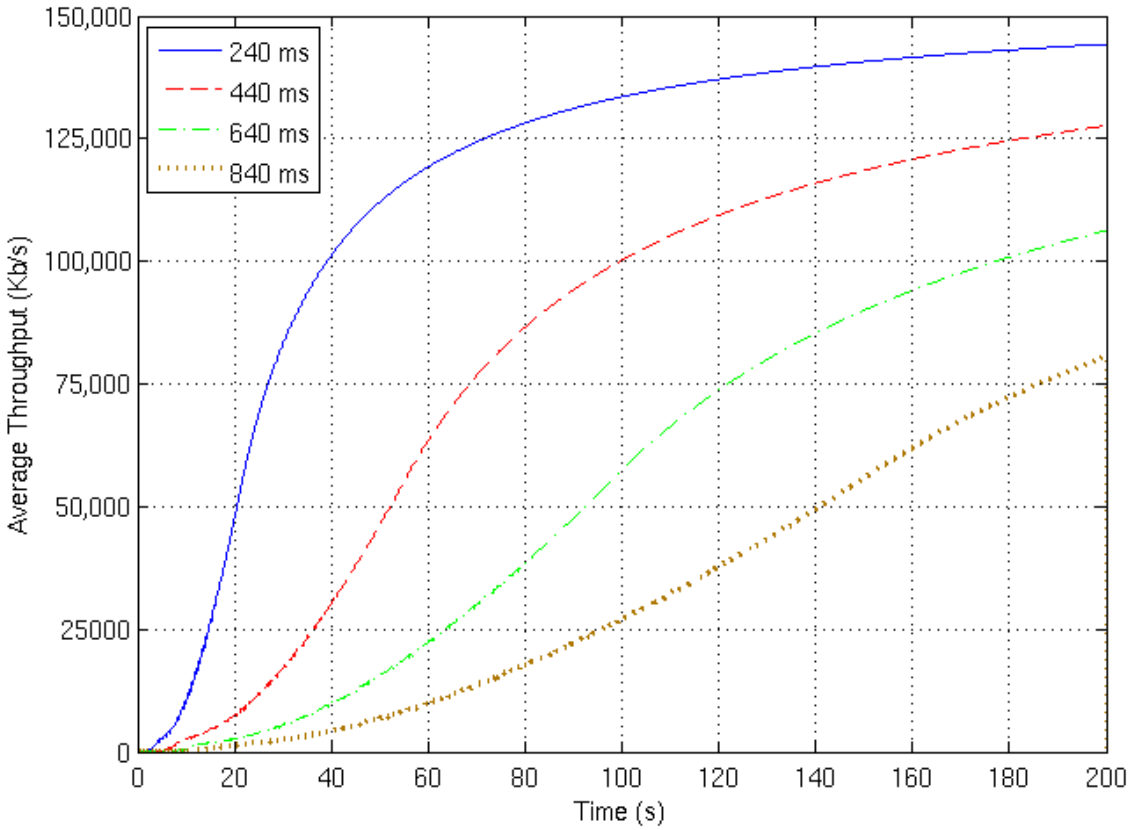

圖 4-11 本文機制在不同 RTT 之平均吞吐量比較圖

由圖 4-11 可得知，當 RTT 時間越短，整體吞吐量的成長速度越快，主要原因是發送端送出封包到收到 ACK 的時間短，調整速率上變比較快速，反之時間越久，則調整速率上就越緩慢，但不論 RTT 大小，所形成的吞吐量形狀皆如圖所示，一開始緩啟動的階段速度緩慢上升，進入本文機制後，吞吐量快速提升，最後慢慢進入平緩階段。

由表 4-3 可看出，RTT 數值越小，在相同的時間內平均吞吐量即越能達到瓶頸鏈路的頻寬上限，反之則數值越低落，主要是由於 RTT 的延遲影響 *cwnd* 的成長速度，雖然如此，在此四種 RTT 數值之下本文機制皆能保持超過一半的瓶頸鏈路的頻寬，而瓶頸頻寬越小，所達到其上限之時間也相對的越短，也可驗證本文所提之機制可正確無誤的應用在不同的 RTT 數值，也即是可應用在不同距離之下的網路情境。

表 4-3 本文機制在不同 RTT 之平均吞吐量比較表

RTT 數值(ms)	平均吞吐量(200 秒內)	達到頻寬吞吐量上限時間(秒)
240	144.17 Mbps	22.13
440	127.48 Mbps	57.29
640	106.07 Mbps	102.52
840	80.35 Mbps	156.35

IV. 為了驗證及分析目標四，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處同樣採用點對點的單一連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，X 值分別設定為 T1、T3、100、155 及 622，Y 值設定為 110，模擬時間設定為 100 秒。

步驟 2：以本文所提出之新機制分別執行上述不同 X 值的模擬環境。

步驟 3：依模擬情形分別記錄其 *cwnd* 成長情況與 RTT 之情況。

步驟 4：計算其平均吞吐量，並作分析比較。

由圖 4-12 可看出，本文機制在不同頻寬之下 *cwnd* 在瓶頸頻寬越小的穩況之下，成長的速度越快，待 *cwnd* 成長至頻寬上限之後，將依本機制在第三區間微調的機制慢慢將 RTT 減少，導致 *cwnd* 值隨著 RTT 減少而跟著減少，622Mbps 雖然也在作微調，但效果因時間太短而不明顯，其它頻寬則可明顯看出 *cwnd* 值隨著 RTT 的降低跟著降低。由圖 4-13 可清楚的看出不同頻寬之下 RTT 的變化，其中 T1 在 4 秒多時 RTT 達到最高值，主要是因為 *cwnd* 成長數值快速的緣故，再加以貪婪地利用整個緩衝區間的頻寬，故 RTT 起伏變動量很大，而 4 秒多後達到穩態便開始修正 RTT 的大小，最後修正至 270ms 左右，即被本文機制認為目前已達穩

態的 RTT 數值而不再增減。T3 以上的頻寬大小也有如此的情境。

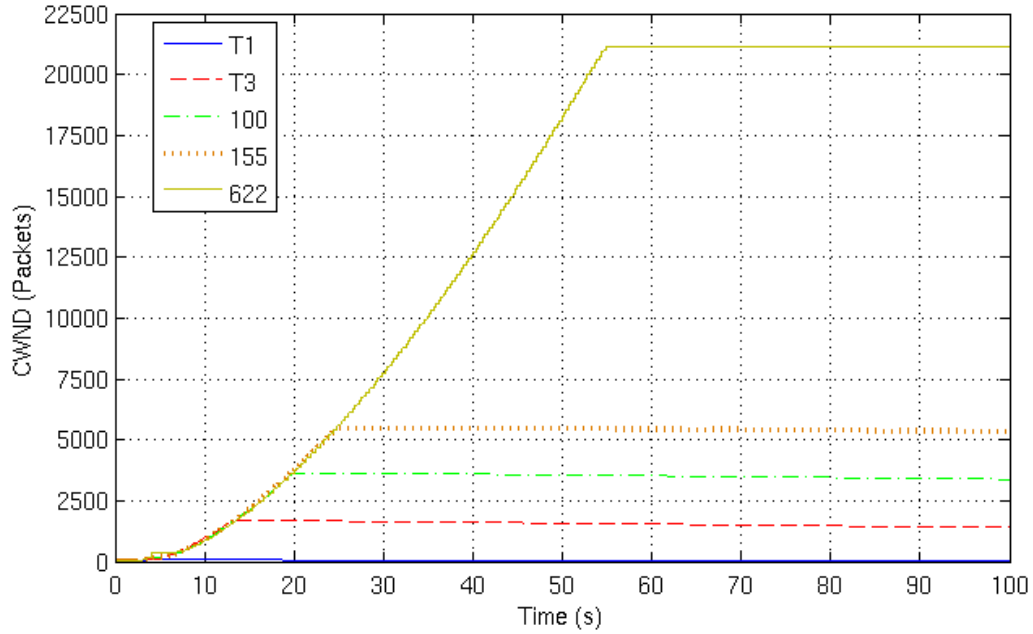

圖 4-12 本文機制在不同頻寬之 cwnd 比較圖

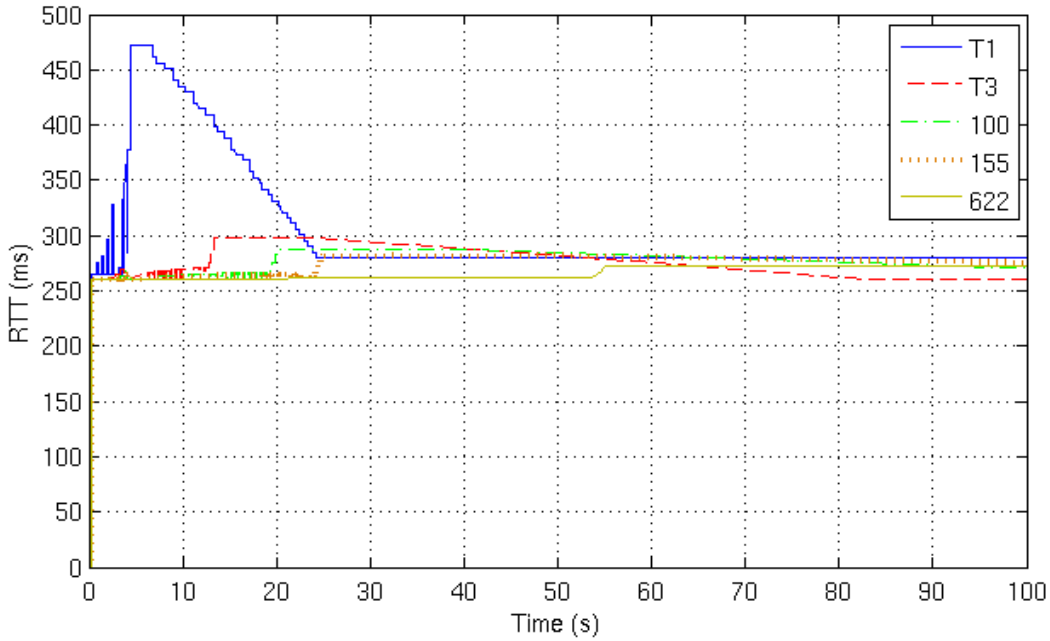

圖 4-13 本文機制在不同頻寬之 RTT 比較圖

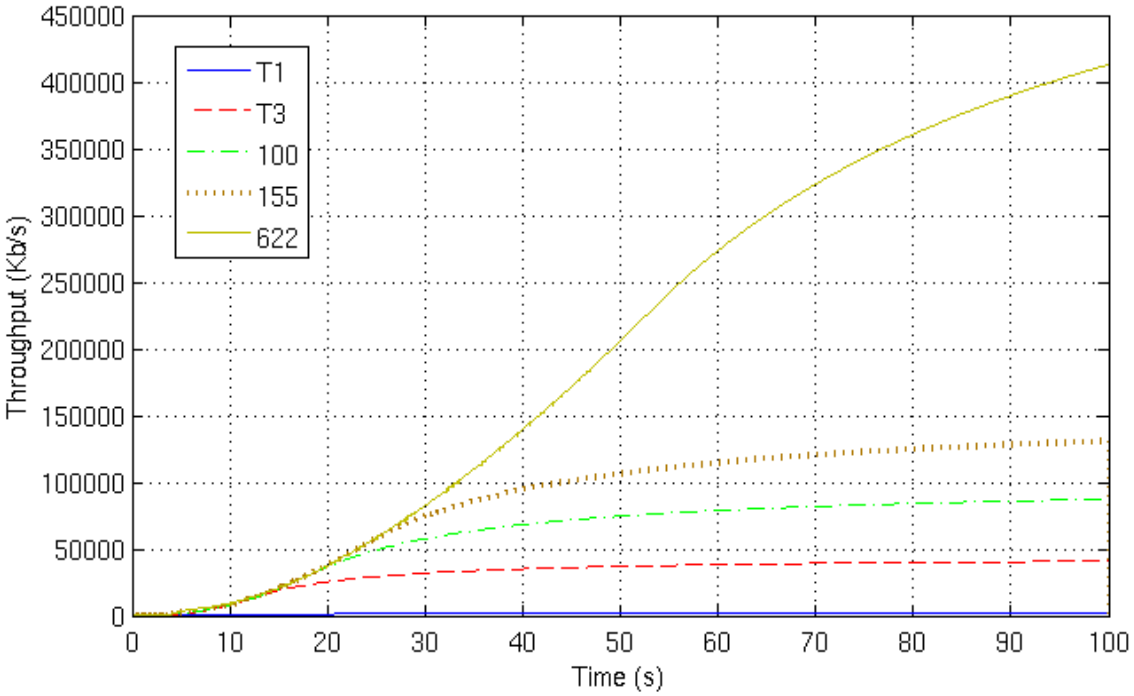

圖 4-14 本文機制在不同頻寬之平均吞吐量比較圖

由圖 4-14 可看出，當可用頻寬越高，本文機制要達到頻寬吞吐量上限的時間就越久，而吞吐量成長圖形與之前吞吐量比較圖均一致，都是一開始經過緩啟動階段快速成長，進入本文機制後吞吐量快速爬升至頻寬上限，接著慢慢趨近平緩直到連線結束。由表 4-4 可看出，頻寬越高，在 100 秒內所能得到的平均吞吐量距離頻寬上限越多，主要是大多數時間用在提升頻寬至頻寬上限，故平均下來將整體的頻寬吞吐量拉低，但不論頻寬高低，皆能達到頻寬上限 155Mbps，差別在於達到的時間長短，由此實驗可驗證本文機制不論頻寬高低皆可正確的應用於不同的網路情境。

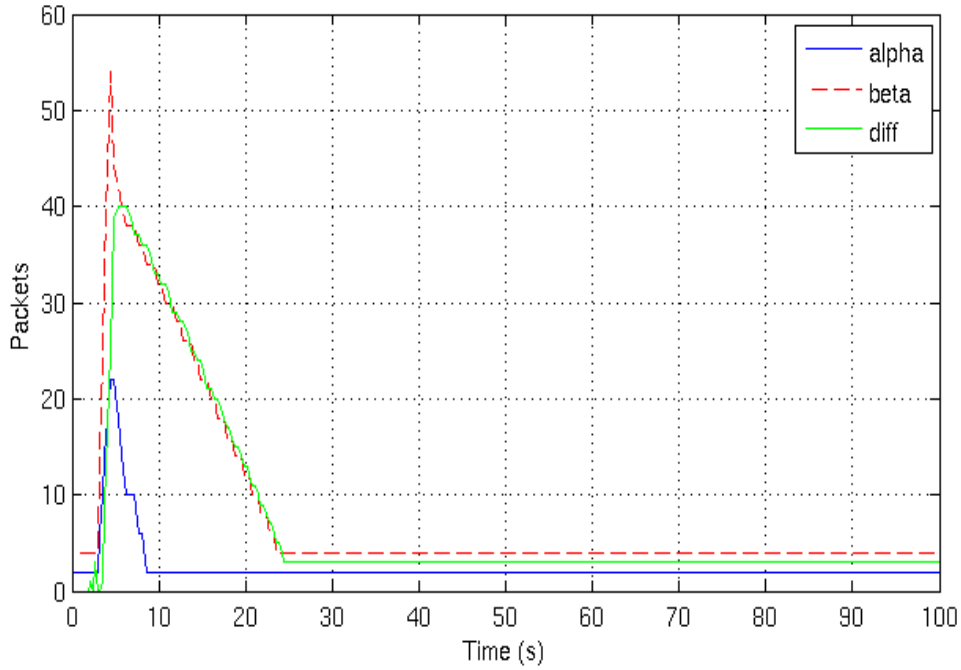

圖 4-15 本文機制在 T1 之 α 、 β 及 *diff* 成長圖

由圖 4-15 可看出，本文機制應用在 T1 時，由於 cwnd 成長速度快速，應用在 T1 時立即達到頻寬上限，故在大約在 4 秒左右即不斷成長至第三區間，在 6 秒時即在第四區間與第三區間不斷來回，直到 24 秒左右達到穩定的數值而不再做調整。

表 4-4 本文機制在不同頻寬之吞吐量比較表

頻寬大小(Mbps)	**平均吞吐量(100 秒內)**	**達到瓶頸頻寬上限時間(秒)**
T1	1.50 Mbps	4.02
T3	40.83 Mbps	13.04
155	130.60 Mbps	24.74
622	412.32 Mbps	54.90

V. 為了驗證及分析目標五，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處採用點對點的單一連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，FTP 設定值為 X 值設定為 622，Y 值設定為 110，另外再加入一條背景流量，此處採用固定流量的 CBR 當背景流量，流量大小為 45Mbps，模擬時間總共設定為 100 秒。

步驟 2：首先啟動本文所提出之新機制，在 60 秒時加入背景流量，而 80 秒時關閉此背景流量。

步驟 3：依模擬情形分別記錄其 *cwnd* 成長情況與 RTT 之情況。

步驟 4：分別計算背景流量出現後本文機制穩態後之 FTP 與 CBR 的吞吐量，並作分析比較。

步驟 5：將背景流量分別增大到 100Mbps、155Mbps，再回到步驟 3、4 作分析比較。

步驟 6：將背景流量修正為從 20 秒開始執行，每隔 20 秒增加另一背景流量，直到 80 秒，總共另外再新增三筆背景流量，流量大小皆為 45Mbps，再回到步驟 3、4。

由圖 4-16 可看出，在 60 秒時開啟 CBR 背景流量後，本文所提之機制立刻因 RTT 的上升而減少 *cwnd* 數值，整個調整時間大約花費 1 秒左右即調整完畢重新進入穩態，待 80 秒關掉 CBR 背景流量後，本文機制因 RTT 的下降而提升 *cwnd* 數值，直到原本的認定最佳穩態時的 *cwnd* 值。而由圖 4-17 可與圖 4-16 作一比對，本文機制在 55 秒時進入頻寬上限，而在 60 秒增加背景流量之後，RTT 瞬間升高，使得本文機制速入區間四而減少 *cwnd* 數值，之後重新進入穩態後 RTT 比原本單一連線時稍高，待 80 秒關掉背景流量後，RTT 瞬間下降，於是本文機制進入了區間一的情況，開始提升 *cwnd* 數值，直到原本穩態的 RTT 大小。

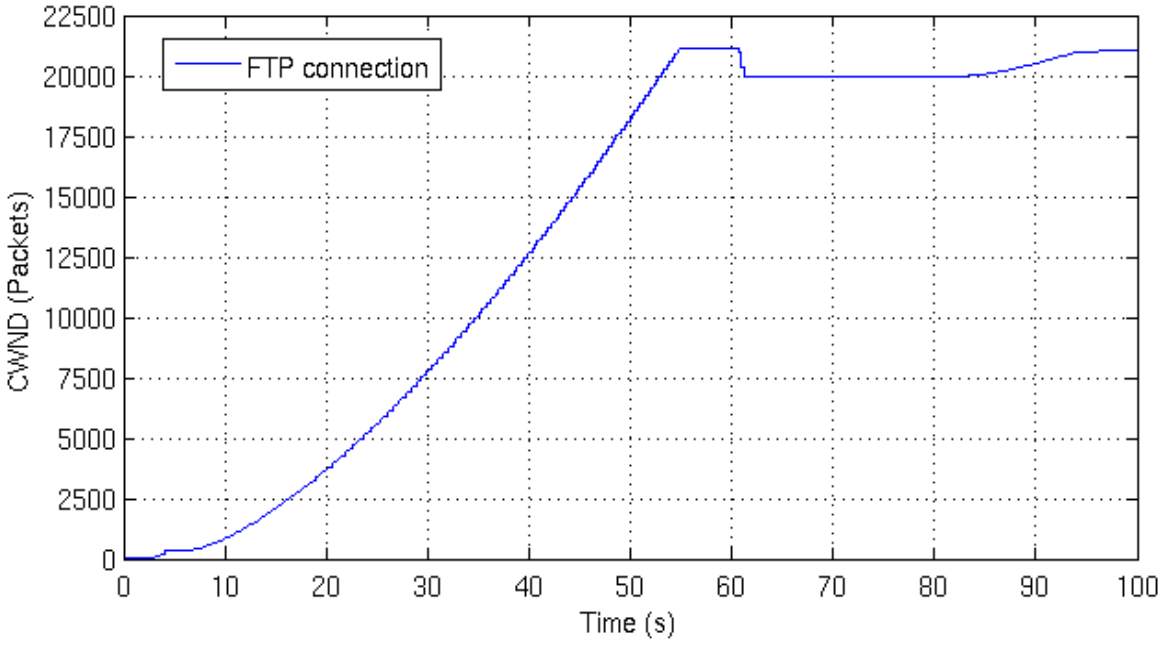

圖 4-16 本文機制在固定 CBR 背景流量(45Mbps)之 cwnd 圖

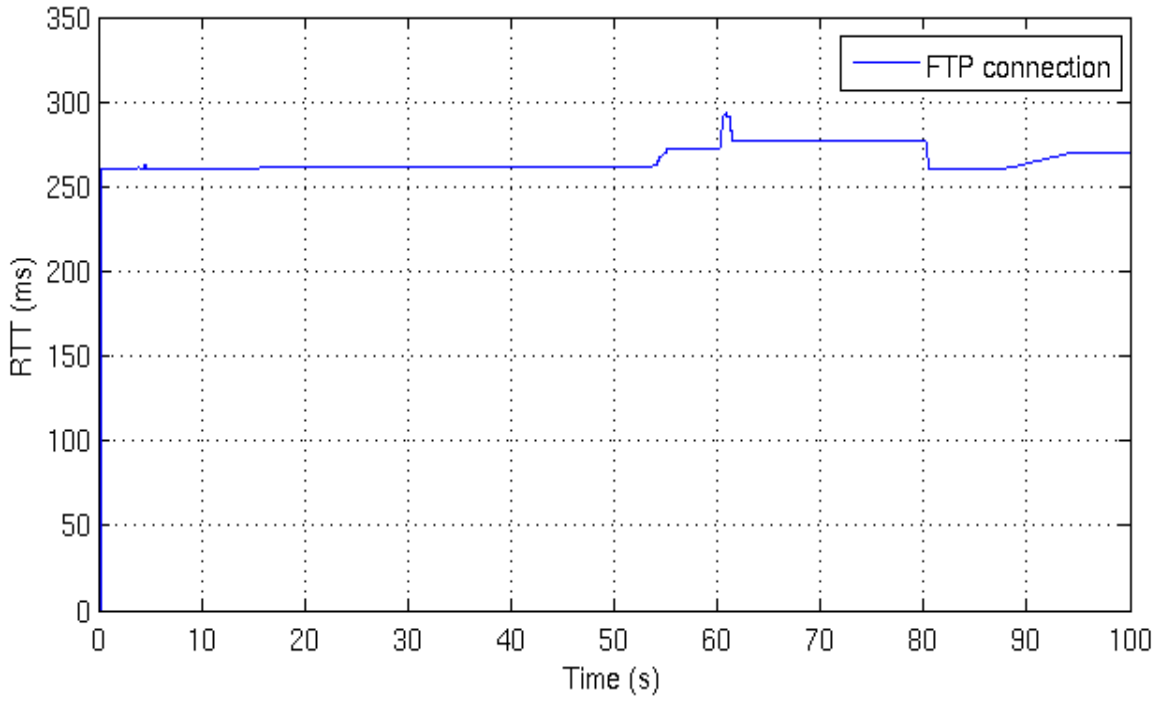

圖 4-17 本文機制在固定 CBR 背景流量(45Mbps)之 RTT 圖

由表 4-5 可看出，本文機制在新加入背景流量後，由原本達到 622Mbps 的頻寬大小，減少至 577Mbps，20 秒內平均數值為 577.33Mbps，而減少的頻寬也確實的應用在 CBR 背景流量，讓 CBR 頻寬達到 45Mbps，平均數值為 44.67Mbps。

表 4-5 本文機制在固定 CBR 背景流量(45Mbps)之平均吞吐量比較表

流量名稱	平均吞吐量(20 秒內)
FTP	577.33 Mbps
CBR	44.67 Mbps

由圖 4-18 可看出，本文機制在 60 秒加入一條 100Mbps 的 CBR 背景流量後，會迅速的減少 *cwnd* 以分配給 CBR 流量使用，大約花了 1 秒左右即降至最低點，但減少完之後本機制發現網路尚有可用頻寬，原因是本機制在釋放資源出來時，並非只有釋放 CBR 所需的頻寬，而是釋放過多的頻寬因而造成尚有多餘的頻寬未被使用，故在 65 秒左右本機制開始提升 *cwnd* 以增加吞吐量，至 75 秒時達到穩態，之後 80 秒關閉 CBR 流量，本機制則開始提升整個網路頻寬回到瓶頸鏈路的頻寬網路速率 622Mbps。由圖 4-19 可對照出，60 秒時 RTT 大增，故本文機制快速減少 *cwnd* 值以因應網路情況，待稍微穩定後，發現仍有可用頻寬，故在 65 秒左右又開始提升 *cwnd* 值而造成 RTT 的緩慢上升，待 75 秒左右進入穩態後即不再增加 RTT，等到 80 秒關閉 CBR 流量之後，RTT 快速下降，本文機制又開始增加 *cwnd*，直到回到原本瓶頸鏈路的頻寬為止。

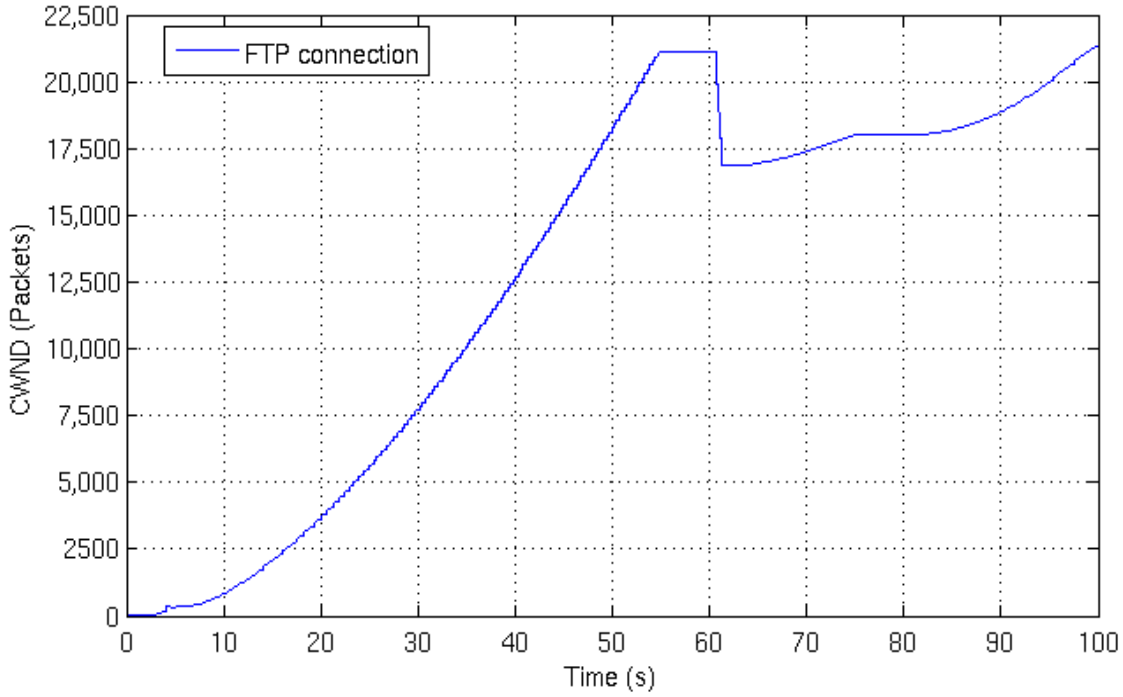

圖 4-18 本文機制在固定 CBR 背景流量(100Mbps)之 cwnd 圖

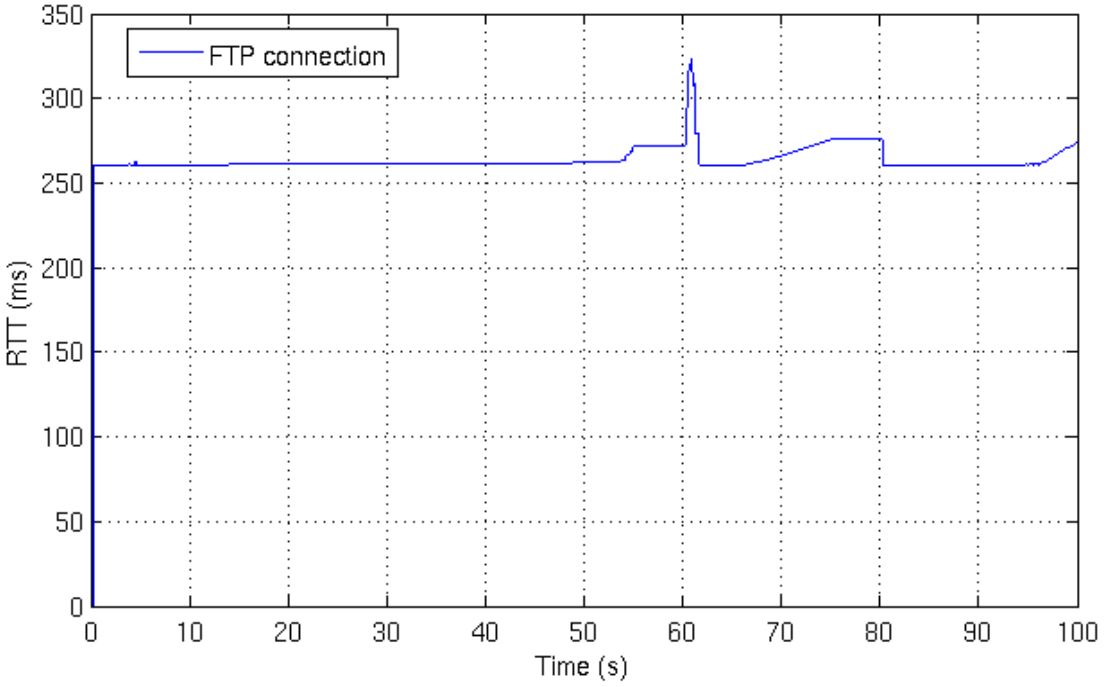

圖 4-19 本文機制在固定 CBR 背景流量(100Mbps)之 RTT 圖

由表 4-6 可看出，本文機制在新加入背景流量後，由原本達到 622Mbps 的頻寬大小快速的減少，在 20 秒內平均吞吐量數值為 522.22Mbps，而減少的頻寬大部分也確實的應用在 CBR 背景流量，讓 CBR 頻寬達到 100Mbps，平均數值為 99.27Mbps，因為加入 CBR 流量而造成本文機制釋放太多的頻寬在之後也確實的被本文機制所應用。

表 4-6 本文機制在固定 CBR 背景流量(100Mbps)之平均吞吐量比較表

流量名稱	平均吞吐量(20 秒內)
FTP	522.22 Mbps
CBR	99.27 Mbps

由圖 4-20 可看出，在 CBR 背景流量增加至 155Mbps 後，其整體 *cwnd* 大致成長如 100Mbps 時一樣，都是快速減少 *cwnd* 後，再慢慢提升回頻寬上限，待移除 CBR 背景流量後，即提升回原本的 *cwnd* 數值。再由圖 4-21 比對得知，最後由於尚未回到原本的 *cwnd* 數值，故 RTT 並未如之前一樣增加，但在 88 秒之後 RTT 有些微變動，主要是在區間二時緩慢增加 *cwnd* 的緣故。

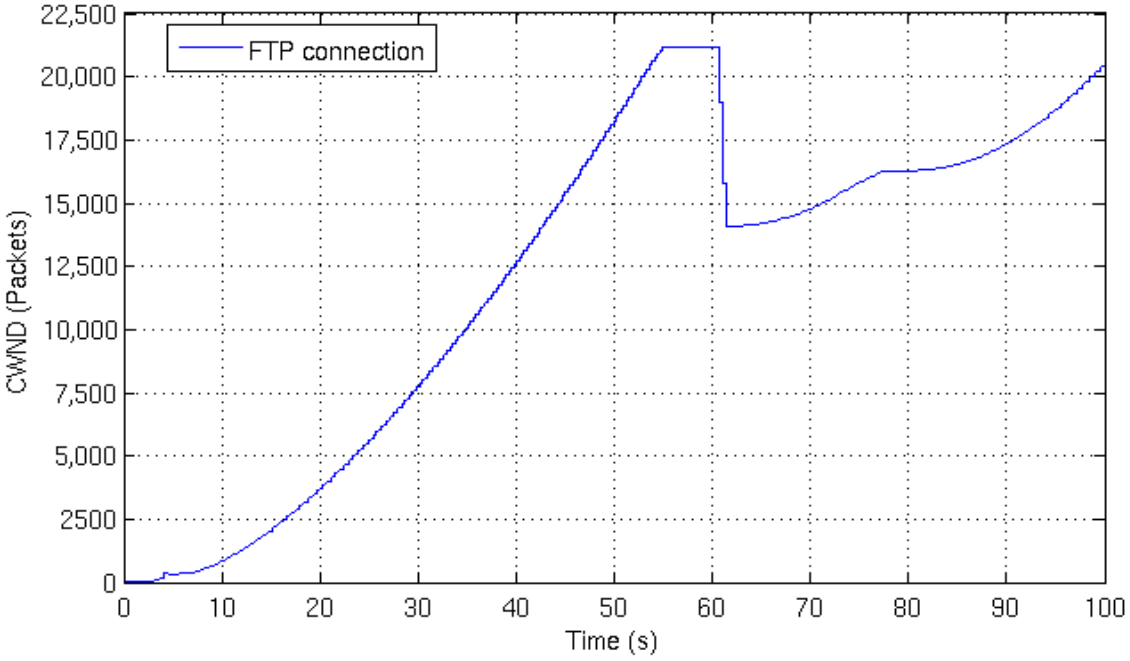

圖 4-20 本文機制在固定 CBR 背景流量(155Mbps)之 RTT 圖

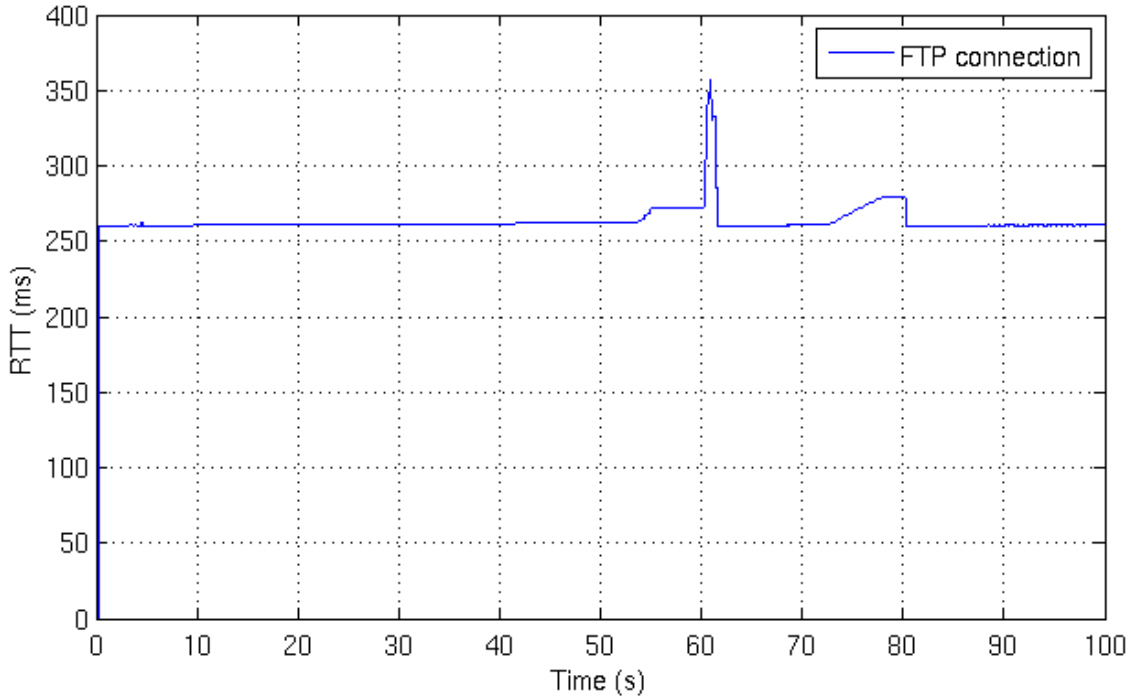

圖 4-21 本文機制在固定 CBR 背景流量(155Mbps)之 RTT 圖

由表 4-7 可看出，本文機制在新加入背景流量後，由原本達到 622Mbps 的頻寬大小快速的減少，在 20 秒內平均吞吐量數值為 455.67Mbps，而減少的頻寬如同 CBR 頻寬 100Mbps 時一樣，大部分也確實的應用在 CBR 背景流量，讓 CBR 頻寬達到 155Mbps，平均數值為 153.85Mbps，因為加入 CBR 流量而造成本文機制釋放太多的頻寬在之後也確實的被本文機制所應用。

表 4-7 本文機制在固定 CBR 背景流量(155Mbps)之平均吞吐量比較表

流量名稱	平均吞吐量(20 秒內)
FTP	455.67 Mbps
CBR	153.85 Mbps

由圖 4-22 可看出，本文機制在 20 秒時加入 CBR 背景流量，因為在 20 秒時，本文機制並未達到此網路頻寬的上限，故從圖中無法看出任何變化，而 40 秒時又新增一條 CBR 流量，因同樣未達頻寬上限故同樣看不出變化，直到 50 秒後達到頻寬上限後，即呈現穩態的狀態，到 60 秒又新增一條 CBR 流量，此時本文機制則減少 *cwnd* 值以釋放網路頻寬給新的 CBR 流量使用，同樣到 80 秒時又新增一條

CBR 流量，故本文機制再次釋放 *cwnd* 值給新流量使用，之後大約在 85 秒因本文機制釋放過多的頻寬而在此時增加 *cwnd* 值以使用尚未用到的頻寬，至 94 秒左右達到最後的穩態。

再由圖 4-23 可看出，每當有新的 CBR 流量連線，RTT 均會突然增高，隨即在本文機制處理之下，立刻減少其 RTT 數值，如 40 秒、60 秒及 80 秒皆是明顯的例子，而當 RTT 減少後，本文機制因發覺網路尚有可用頻寬，即是剛釋放太多頻寬，故又開始增加 *cwnd* 數值，直到頻寬達到上限，故 RTT 在減少之後又隨之增加一些，主要原因在此。

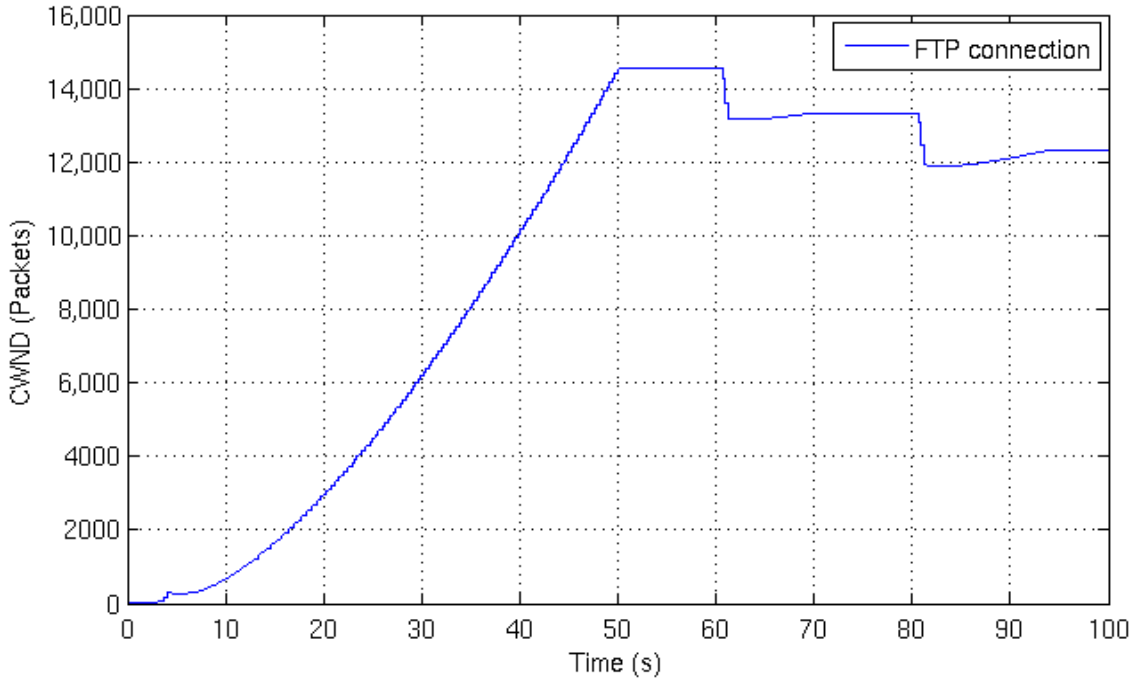

圖 4-22 本文機制在固定多條 CBR 背景流量(45Mbps)之 cwnd 圖

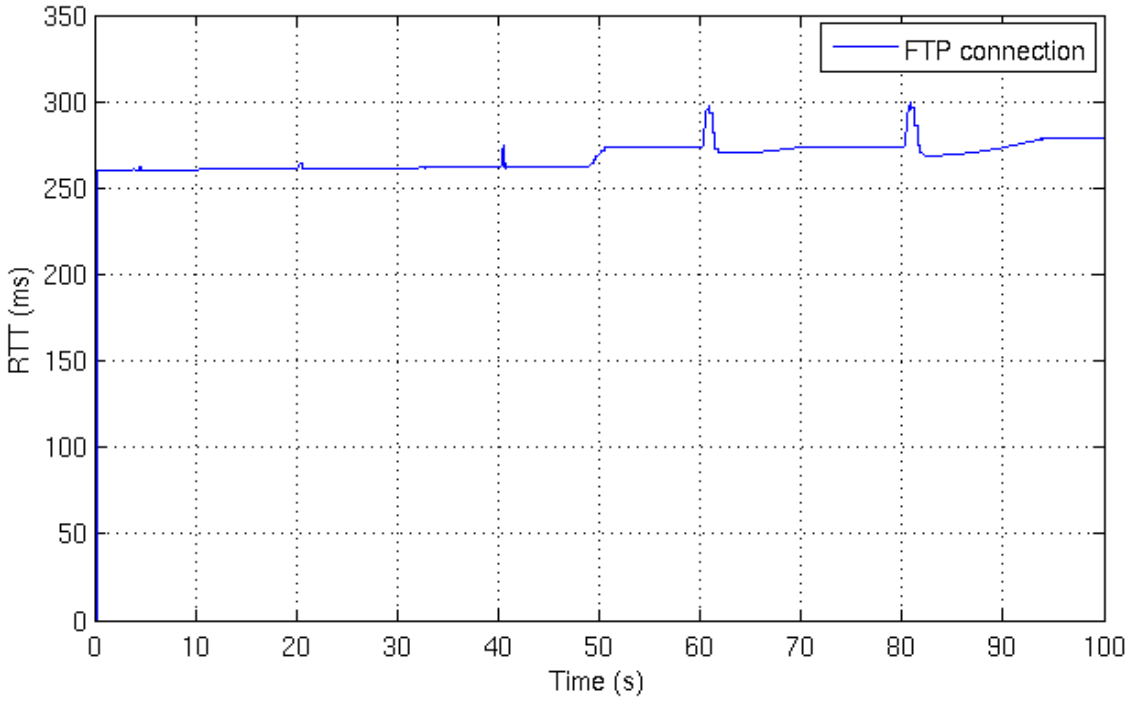

圖 4-23 本文機制在固定多條 CBR 背景流量(45Mbps)之 RTT 圖

由表 4-8 可得知，在 20 秒至 40 秒內，由於本文機制尚在提升吞吐量並未達到頻寬上限，故在此段時間內只達到 239.86Mbps 而已，而 20 秒加入的 CBR 流量則平均達到 44.70Mbps；而在 40 秒至 60 秒內，本文機制雖然在 50 秒左右達到頻寬上限，但因 40 秒加入的新 CBR 流量，造成本文機制減少 *cwnd* 值以釋放頻寬給新的 CBR 流量使用，故造成在 40-60 秒內本文機制達到 498.83Mbps，而兩條 CBR 流量分別有 44.97Mbps 與 44.68Mbps 的頻寬，總和頻寬尚未達 622Mbps 主要是因為在 40 秒至 50 秒之間本文機制尚未達到頻寬上限而在提升吞吐量中，故平均下來總和無法達到頻寬上限 622Mbps；至於 60 秒至 80 秒及 80 秒至 100 秒，本文機制也確實的將頻寬分享給新加入的 CBR 流量，總和網路頻寬也都達到 622Mbps。經由上述不同背景流量情境之模擬實驗得知，本文機制在加入新的流量時，可因應不同流量而釋放該流量所必須的頻寬大小，而不會獨占整個頻寬無法容忍其它頻寬的加入。

表 4-8 本文機制在固定多條 CBR 背景流量(45Mbps)之平均吞吐量比較表

流量名稱	平均吞吐量 (20-40 秒內)	平均吞吐量 (40-60 秒內)	平均吞吐量 (60-80 秒內)	平均吞吐量 (80-100 秒內)
FTP	239.86 Mbps	498.83 Mbps	487.32 Mbps	442.36 Mbps
CBR1	44.70 Mbps	44.97 Mbps	45.00 Mbps	44.99 Mbps
CBR2	—	44.68 Mbps	45.00 Mbps	44.99 Mbps
CBR3	—	—	44.68 Mbps	44.99 Mbps
CBR4	—	—	—	44.67 Mbps
總和	284.56 Mbps	588.48 Mbps	622.00 Mbps	622.00 Mbps

VI. 為了驗證及分析目標六，透過 NS2 網路模擬工具，在此設計了整個實驗流程：

步驟 1：建立如圖 4-1 的網路拓樸圖，而此處採用多條的點對點連線方式，以檔案傳輸控制協定(FTP)作資料傳輸，FTP 設定值為 X 值設定為 622，Y 值設定為 110，每條連線的傳輸頻寬皆一樣，模擬時間總共設定為 100 秒。

步驟 2：分別同時建立 2、4、6、8、10 及 12 條的連線作測試，並在連線啟動時，即開啟所有連線傳輸。

步驟 3：依模擬情形分別記錄每條連線情況。

步驟 4：分別計算每條連線的吞吐量，並以式(21)作公平指數的計算後作分析比較。

為測試本文機制之公平性如何，採用由 Jain[19]所提供的公平性指數算式作驗證，公式如式(21)所示：

$$Fairness\_index = \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n\sum_{i=1}^{n} X_i^{\,2}} \tag{21}$$

公式中，n代表全部的連線數目，X_i則是第i條連線的吞吐量，公平性指數由全部連線的吞吐量總和的平方除以連線數目乘上每條連線的平方和的結果，結果越接近 1 代表越公平，反之則越不公平。

由表 4-9 可看出，連線數目越多所得到的平均吞吐量則越高，這是由於大家分配到的頻寬上限減低，故本文機制可較快速的達到頻寬上限，平均起來因此較高，另外可看出本文機制在不同的連線數目之下，均可達到 99%多的公平性。

表 4-9 本文機制在多條連線之公平指數比較表

連線數目	平均吞吐量	公平性指數(%)
2	250.98 Mbps	99.98
4	137.10 Mbps	99.96
6	94.00 Mbps	99.91
8	71.27 Mbps	99.96
10	57.39 Mbps	99.74
12	48.10 Mbps	99.34

由表 4-10 與表 4-11 可得知本文機制在異質網路之下，即與 TCP New Reno 機制做公平的網路頻寬競爭，可達到 92.74%的公平指數，反觀 TCP Vegas 機制則只能達到 54.26%的公平指數，相比之下本文機制較原本 TCP Vegas 機制公平性更佳。

表 4-10 本文機制與 TCP New Reno 公平性指數比較表

連線機制名稱	平均吞吐量(100秒內)	公平性指數(%)
New mechanism	320.95 Mbps	92.74
TCP New Reno	180.62 Mbps	

表 4-11 TCP New Reno 與 TCP Vegas 公平性指數比較表

連線機制名稱	平均吞吐量(100秒內)	公平性指數(%)
TCP New Reno	185.98 Mbps	54.26
TCP Vegas	7.94 Mbps	

五、結論與未來展望

本篇論文中，主要應用在長距離高速傳輸網路的 TCP 連線，主要透過 RTT 的量測，再依預期傳送吞吐量跟實際傳送吞吐量差值比對 α 、 β 數值後，評估目前網路情況，依本文所提之四個區間調整不同網路情況時的 *cwnd* 數值，再動態調整 α 與 β 數值，以作更細微的調整。

經由網路模擬實驗一與實驗二可驗證本文所提之機制不但可確實有效的改善目前 TCP Vegas 機制的效能，也比目前改善 TCP Vegas 機制效能的其它幾種方式更佳，更縮短了原本 TCP Vegas 機制達到高吞吐量所需的時間；從模擬實驗三與實驗四中可驗證本文所提之機制不會受到距離的長短或是網路頻寬大小的不同而造成本文機制無法應用於該網路情境；且由模擬實驗五可得知，本文所提之機制不會因突然加入的流量便失去其機制效用，也會釋放新流量所需的網路頻寬給其使用，可見本文機制不會有獨占頻寬的特性；而模擬實驗六得證本文機制之公平性平均可達 99%以上，且較原本 TCP Vegas 機制與 TCP New Reno 機制在頻寬分配上更佳，足見本文機制對於多條連線的頻寬分配有一定的可靠性。

未來的研究工作中，大致上分為以下幾點：

1. 希望能改善本文機制達到頻寬上限後 RTT 增加問題。
2. 達到頻寬上限的時間希望能更加快速。
3. 對於路由器間緩衝區大小的設置問題，期待能有更佳的解決方案。
4. 當連線為進入穩態後調整 α、β 數值以降低 RTT 數值的速度希望能再更快速地調整完畢

因此若能改善上述問題，即能讓本機制更為完善。

參考文獻

[1] 台灣網路中心，網消會，資策會 FIND/經濟部技術處「創新資訊應用研究計畫」，"2008 年第一季台灣網際網路連線頻寬調查"，資策會 FIND 網站, 2008.

[2] S. Floyd, T. Henderson, A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC3782, 2004.

[3] L. Brakmo and L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communication, vol. 13, No. 8, pp.1465-1480, 1995.

[4] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP Vegas: Emulation and experiment", SIGCOMM, pp.185-205, 1995.

[5] 說明 TCP Flow control 機制, http://wiki.iiietc.ncu.edu.tw/mediawiki/index.php?title=%E8%AA%AA%E6%98%8E_TCP_Flow_control_%E6%A9%9F%E5%88%B6%3F.

[6] J. Semke, J. Mahdavi, M. Mathis, "Automatic TCP Buffer Tuning", the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication, pp.315-323, 1998.

[7] B. L. Tierney, D. Gunter, J. Lee, M. Stoufer, J. B. Evans, "Enabling Network-Aware Applications", 10th IEEE International Symposium on High Performance Distributed Computing, pp.0281-0288, 2001.

[8] T. Dunigan, M. Mathis, B. Tierney, "A TCP Tuning Daemon", the ACM/IEEE conference on Supercomputing, pp.9-24, 2002.

[9] S. Floyd, "HighSpeed TCP for Large Congestion Windows", RFC 3649, 2003.

[10] D. X. Wei, C. Jin, S. H. Low, S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance", IEEE/ACM Transactions on Networking, vol. 14, pp.1246-1259, 2006.

[11] T. Kelly, "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks", ACM SIGCOMM Computer Communication Review, vol. 33, pp.83-91, 2003.

[12] R. N. Shorten, D. J. Leith, J. Foy, R. Kilduff, "Analysis and design of congestion control in synchronized communication networks", the 12th Yale Workshop on Adaptive and Learning Systems, 2003.

[13] Y. C. Chan, C. T. Chan, and Y. C. Chen, **"**An enhanced congestion avoidance mechanism for TCP Vegas", Communications Letters, IEEE, vol. 7, pp.343-345, 2003.

[14] H. Zhou, “STT-Vegas: A Simple Single-Trip Time Based Modification of Vegas”, Proceedings of IEEE Consumer Communications and Networking Conference, vol. 191, No. 25, pp.453-457, 2006.

[15] A. Maor and Y. Mansour, “Ada Vegas: Adaptive Control for TCP Vegas”, Global Telecommunications Conference, IEEE , vol. 7, pp.3647-3651, 2003.

[16] K. N. Srijith*, L. Jacob and A. L. Ananda, “TCP Vegas-A: Improving the Performance of TCP Vegas”, Proceedings of SCI Computer Communications, vol. 28, Issue. 4, pp.429-440, 2004.

[17] Y. C. Chan, C. L. Lin, C. T. Chan and C. Y. Ho, “Improving Performance of TCP Vegas for High Bandwidth-Delay Product Networks”, Proceedings of IEEE Advanced Communication Technology, vol. 1, pp.987-997, 2006.

[18] NS2, Network Simulator version 2, http://www.isi.edu/nsnam/

[19] Jain’s Fairness Index, http://www.cs.berkeley.edu/~kfall/EE122/lec21/sld005.htm