**專題報告**

**Reliable In-Network Aggregation**

姓名：王敬智

學號：109611058

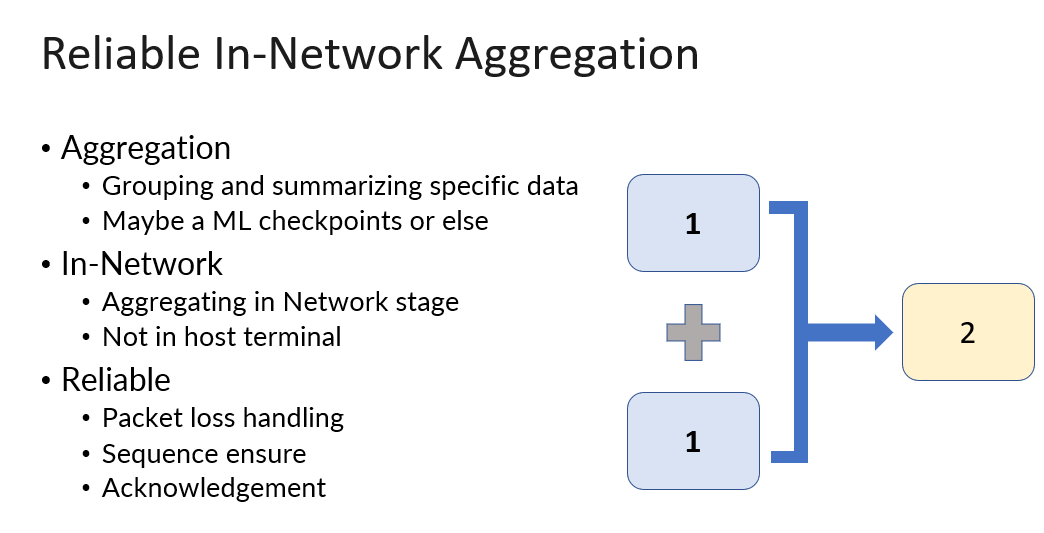
同組組員：吳典謀

指導教授：林靖茹教授

本專題程式碼網址: [https://github.com/LittleD3092/reliable-in-network-aggregation](project%20report.docx)

**Reliable In-Network Aggregation:**

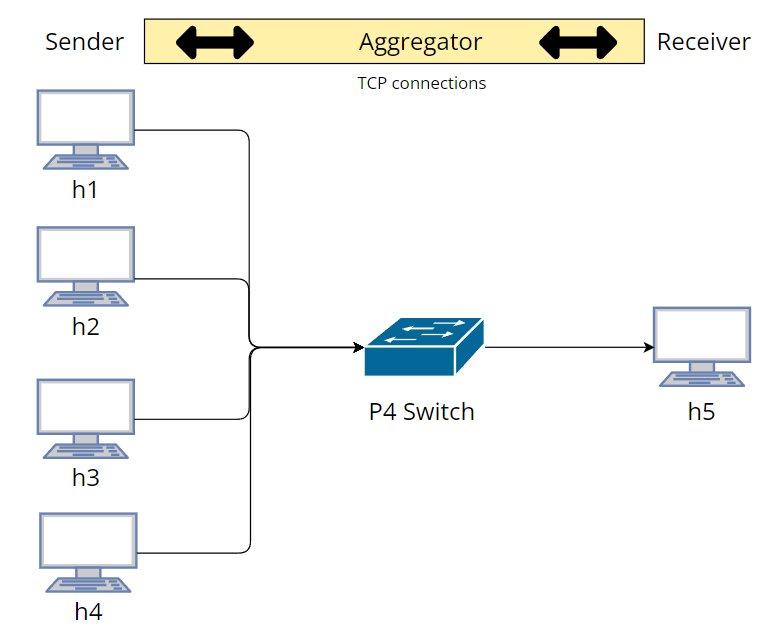
本項專題由三個核心機制組成，即是組成專題名稱的Reliable、In-Network以及Aggregation三個部分。專題內容主要是在P4語言所執行的軟體定義網路(Software-defined Networking)環境下，實作出可程式化交換機可在TCP的連線中運作的資料聚合/融合機制。



**實驗方式：**

實驗利用Mininet模擬網路環境，產生出對應的拓樸，終端之間使用運作P4語言的bmv2網路模擬交換機，形成一個小型的獨立模擬網路。

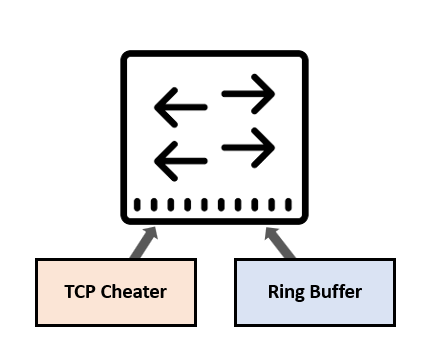
實驗係利用python撰寫之socket程式，於每個sender與receiver間都建立TCP connection，再利用scapy或是iperf來發送封包。封包完成剖析後，switch利用TCP的header來作為辨識封包順序的手段，並將每個封包攜帶的payload存在switch內部的register中。在每個特定序號之payload收集完成後，便會將最終的資訊傳送給接收端的host，因此資料的處理是在網路內完成的，而非傳統的操作終端。



拓樸示意圖

**實作細節：**

實作由兩個機制組成，分別是TCP連線的維持機制以及switch內的資料結構維護方式。TCP屬於RDT，運作機制內會有許多對於封包順序的確保機制，如seq\_num & ack\_num。在實體的可編程交換機上，暫存器的空間通常是有限的，如何確保在有限的空間內，可以對於由網路端不斷傳輸的資料做正確的處理就是設計的重點。



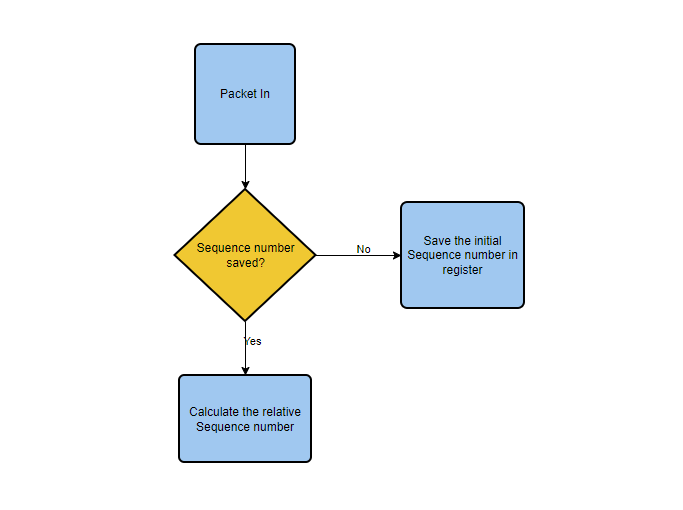
組成示意圖

TCP cheater

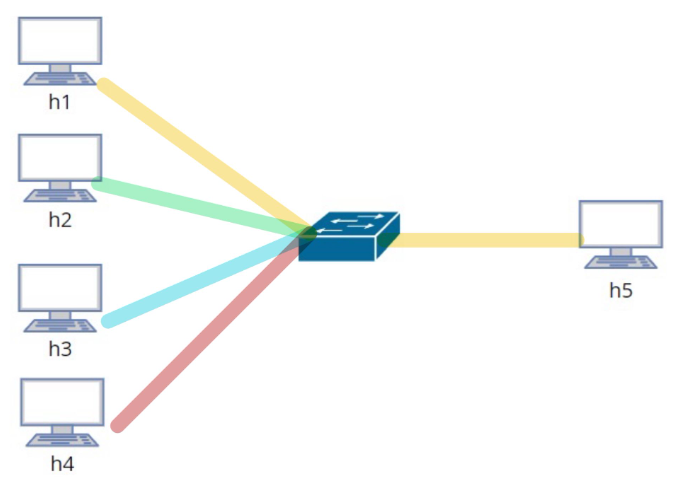
維持TCP連線機制的方法，我們自己稱他為TCP cheater，即透過修改封包內容的方式，欺騙每個終端他的TCP packet是沒有被修改過的。TCP的封包格式是由fixed length header 和option variable length field組成的，因此在parse完fixed length header後，必須使用sub-parser來對可變長度的header做解讀，取出後面重新計算checksum時會用到的option field，不過這邊由於僅需要SACK的部分，因此關閉了timestamp這個option欄位。接著利用TCP以sequence number以及acknowledge number來確保雙方已接收的Byte量這個機制，我們在每個sender以及receiver端完成TCP handshake後，將該TCP connection的initial sequence number以及initial acknowledge number至register內。如此我們控制封包送出時的payload大小為1024 Byte，配合以下公式：

relative\_sum=((initial\_seq\_num-seq\_num)>>10)+1

就掌握每個封包進入switch時，對應於第一個封包的順序，後面我們就可以利用這個順序做payload的處理。

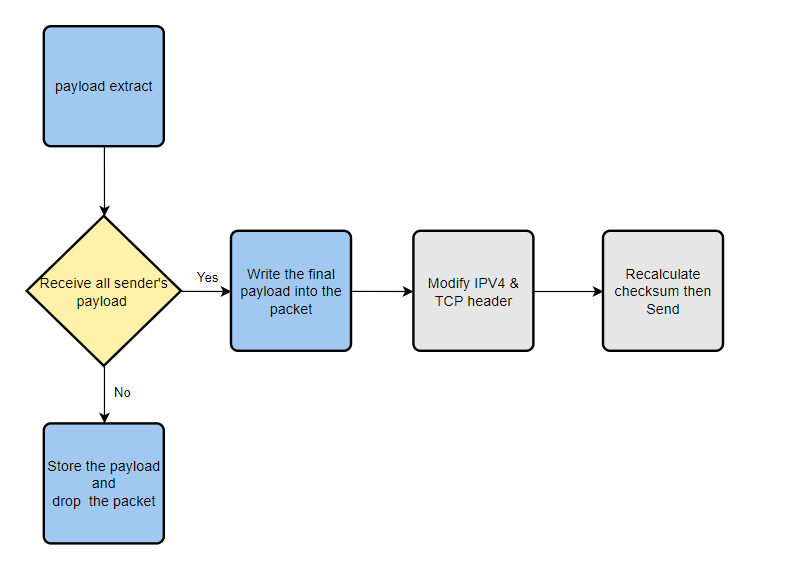


對於封包的處理方式，當資料取出時未滿足傳輸條件時，封包會直接被drop；而當四個發送端的同序號資料都送達switch時，完成加總的資料會被放入最後一個抵達的該序號封包並準備送出。由於前期在parser內已經完成資料的剖析，因此這邊可以直接將最終的結果放入對應的data段內。由於該封包可能來自不同的sender，若是直接送出，可能會使得TCP sequence number錯誤，導致大量錯誤重傳，因此這邊我們選擇其中一個sender做為leader，並以該sender與receiver的TCP connection做為主要的傳輸通道，如下圖所示:

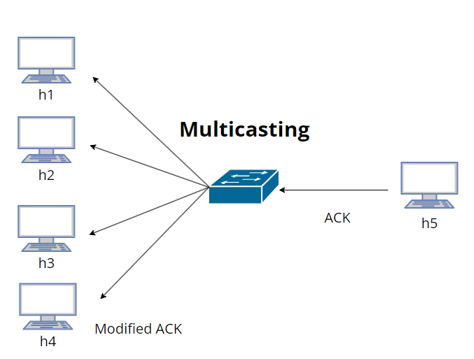


非leader sender的封包會在switch被drop掉，僅有被改為leader sender資訊的封包會被傳送至h5，其餘3個TCP connection由link上看沒有任何流量至h5。

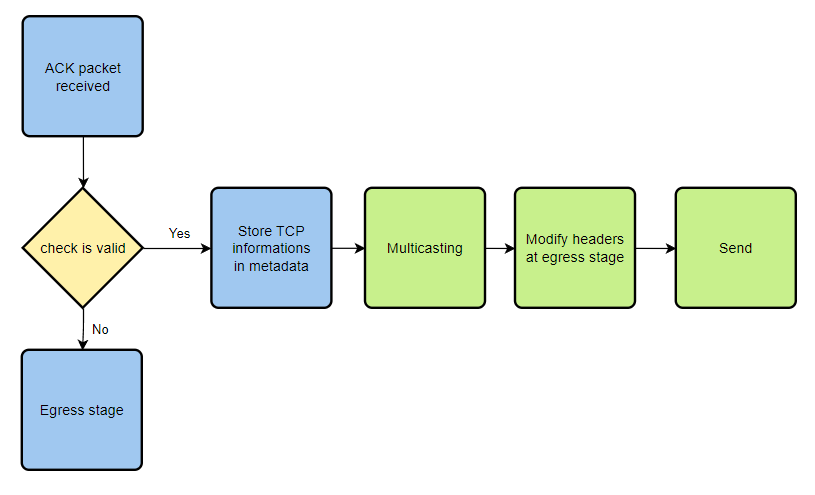
將所有要傳送至receiver的封包header全部改為該sender與receiver的IP、TCP資訊，並重新計算checksum，防止TCP的連線出現錯誤導致大量的重傳封包產生。在最後傳輸前，由於TCP的檢查和checksum機制，因此封包會在重新計算checksum後才送至egress port階段。



在封包傳輸後，由於封包是由leader sender的TCP connection送達的，因此僅會有leader sender收到ACK封包，但在正常的TCP運作情形下，由於其他三個sender也已對receiver送出封包，因此receiver必須回覆所有的sender才能使TCP 重傳停止並繼續傳送。由於receiver實際上只收到來自leader sender的一個封包，因此其他三個sender將無法收到來自receiver的ACK封包。因此，我們使用multicast功能配合條件判斷，在收到來自receiver的ACK封包時，會先計算出該封包的relative\_seq\_num，確認該封包是payload的ACK封包，接著對其使用multicast的功能，將封包由ingress端clone至所有port的egress端。如下圖所示:

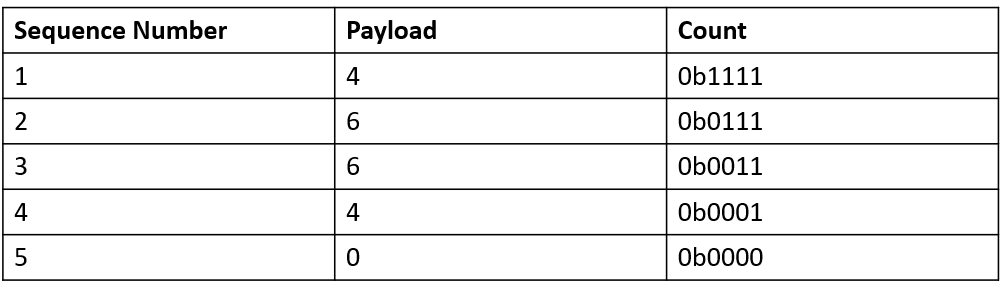


離開ingress stage前，會利用meatadata將每個sender的TCP connection資訊記錄下來並攜帶至egress port，再由egress stage修改封包資訊完成對每個TCP connection的復原以及checksum重新計算，使每個sender都收到對應的ACK封包。



Ring Buffer

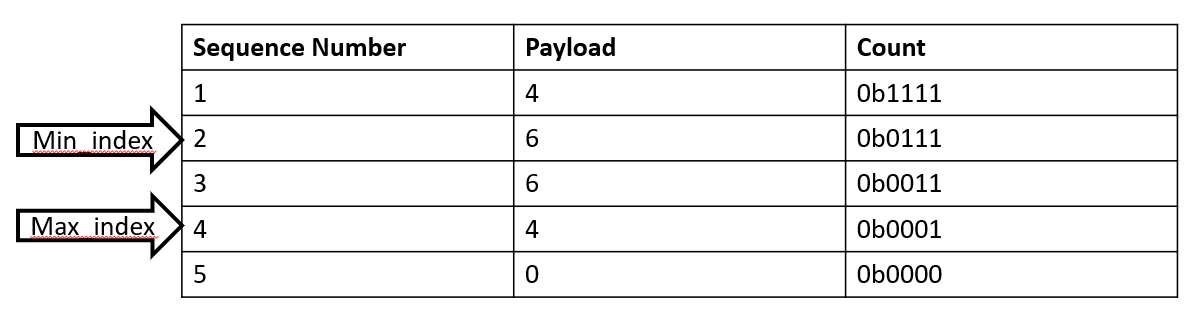
　　為了將發送端的封包所攜帶的資訊聚合，我們需要維護一個資料結構來儲存封包的序號、封包內的payload資料以及目前聚合的進度，此資料結構如下圖所示：



　　Sequence Number欄位會儲存封包的序號，Payload會儲存目前聚合的資料，Count則會以二進位的方式記錄已聚合的發送端，避免重新聚合。

　　由於P4語言與硬體限制無法進行搜尋，Sequence Number在buffer中的排列一定照著順序，使我們可以用序號取餘數得到此序號儲存空間的指標。並且為了記錄buffer的使用情形，我們另外維護了兩個指標min\_index與max\_index，min\_index指向未被ack的最小序號，max\_index則指向最大序號。

　　根據上圖的例子，1號的Count欄位皆為1，代表每個發送端都已聚合過。2到4號的Count有幾個位元為0，代表有幾個發送端未聚合。5號的Count的全部位元為0，代表沒有任何發送端聚合過。因此在上圖的情況下，min\_index指向2號，max\_index指向4號，如下圖。



　　在發送端的封包進到switch時，將序號與min\_index和max\_index對應到的序號比較後有三種情況：

1. 小於min\_index：代表進來的封包已經被聚合過了，因此此封包會直接丟棄。
2. 在min\_index與max\_index之間：代表進來的封包序號正在被聚合，因此可以直接聚合，並更改相對應的Count位元。
3. 大於max\_index：代表進來的封包序號是第一次見到，在檢查有空位後可以把payload存在空位中，並更改max\_index。

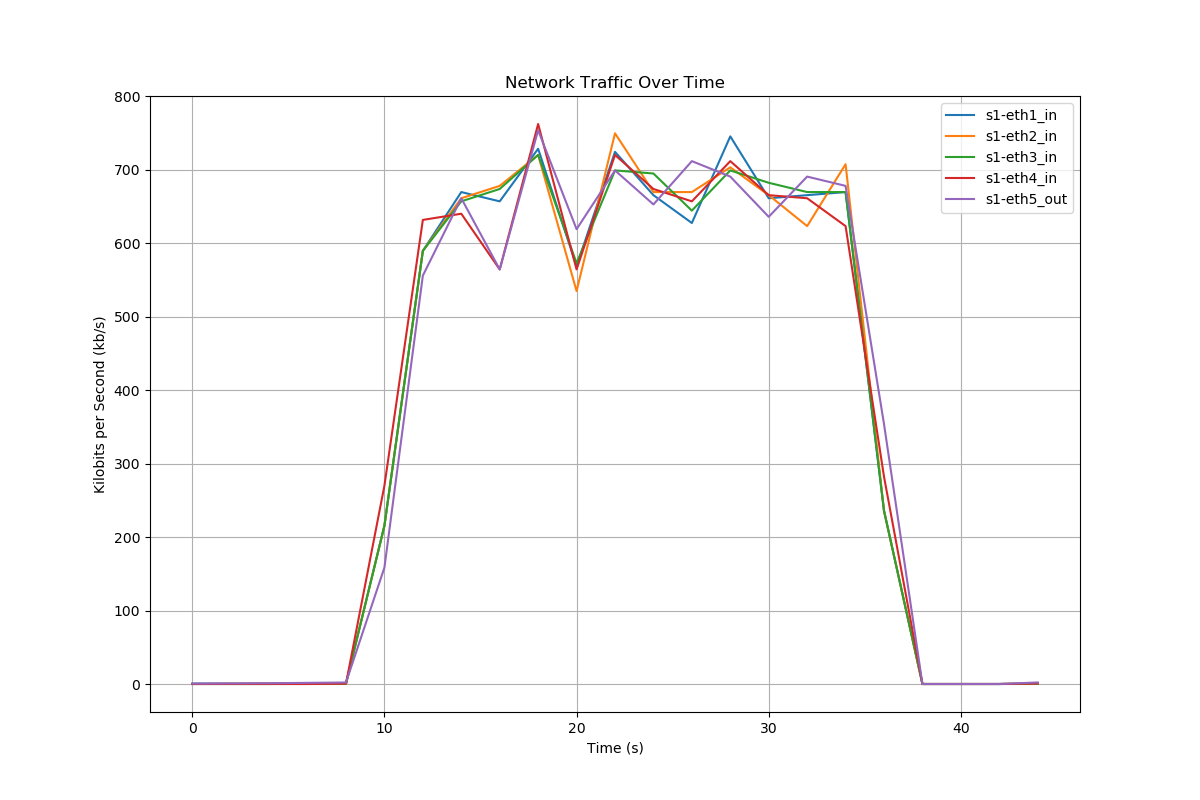
　　在聚合的封包送到接收端後，接收端會回傳ack。在switch接收到ack後會更改min\_index，並把ack廣播到所有的發送端。

**實驗結果**

　　為了測試演算法與實作在不同環境下的結果，使用以下四種環境進行測試：

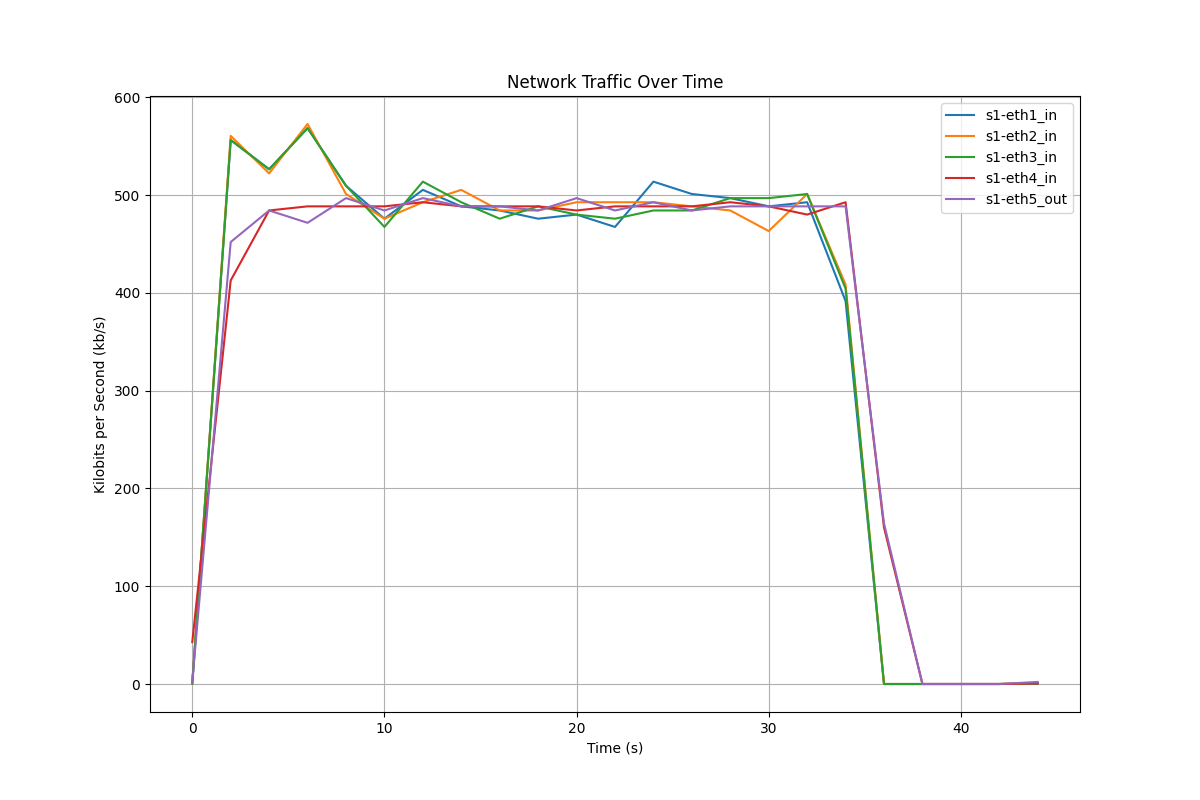
1. Sufficient Bandwidth

所有的link都有足夠的頻寬，並預期每個link都將有差不多的速率。我們將頻寬設定在1Mbps，不過由於模擬環境與硬體效能限制，最高只能跑到700kbps，且傳輸的速率會有較大的波動。由實驗結果圖可以得知，每條link的速度皆是差不多的，與預期的結果幾乎相符。



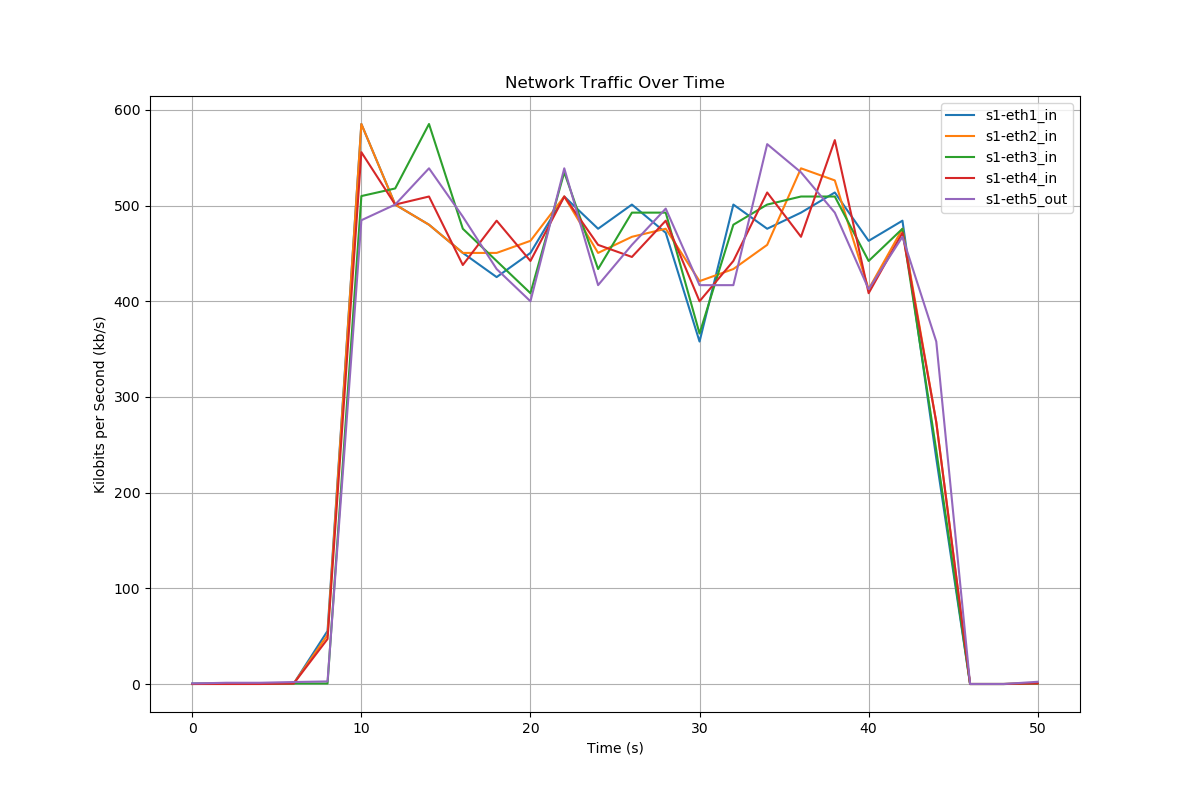
1. Single Slower Sender：

將其中一個發送端的頻寬調較其他三個發送端低。預期結果是速率將因為TCP的機制收斂至接近較低頻寬的發送端。由實驗結果圖也可以得知，儘管一開始其他發送端速度較快，但運作一陣子後，會收斂至相近的速度。



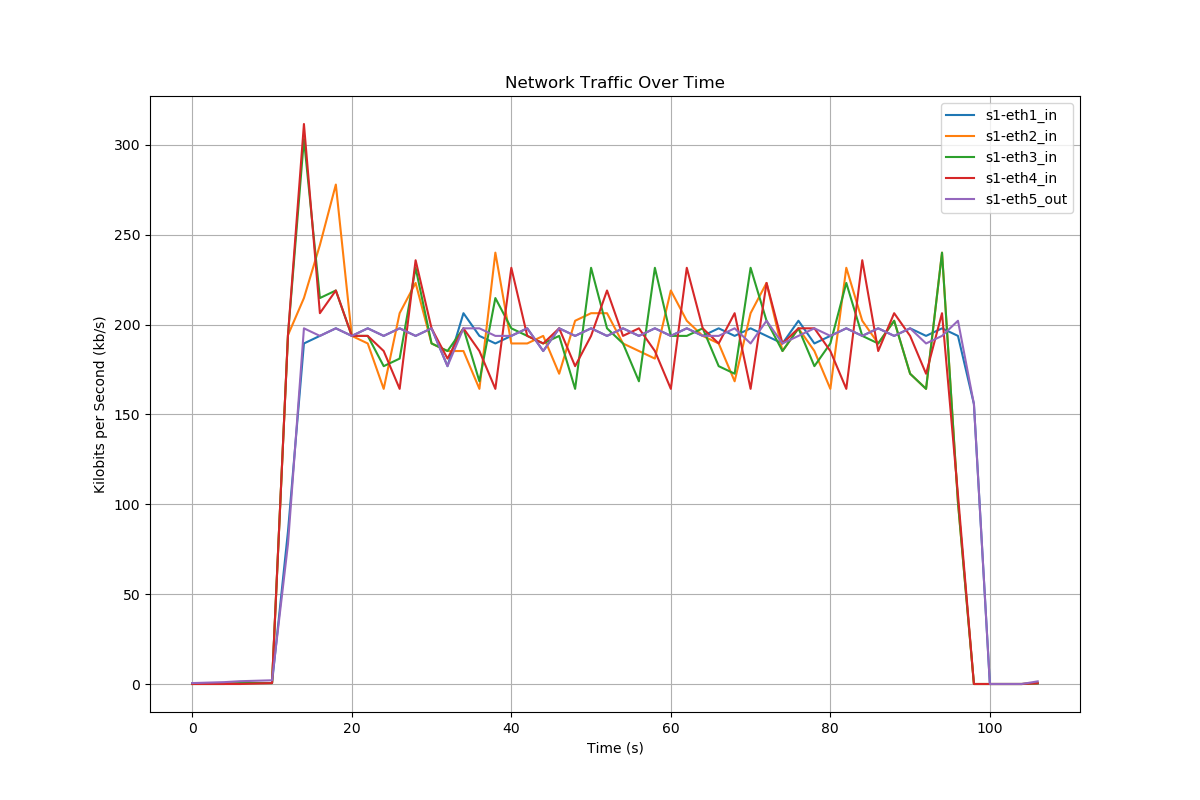
1. Slower Receiver：

接收端的頻寬較低，其餘發送端正常頻寬的情況下，預期瓶頸會發生在switch往接收端的link上，因此速度應收斂至與至接收端相同的速率。在下圖實驗結果中可以看到與第一個情境類似的結果，但由於瓶頸出現在接收端至switch的link上，因此整體速度比第一個情境低。



1. All Different Bandwidth：

所有的發送端與接收端頻寬都不同。可以看到較大頻寬的發送端速度在開頭會較快，之後收斂在最慢的頻寬附近，與我們預期的行為模式相符。



**改進空間:**

實驗結果出來後，我們發現Ring Buffer與TCP的組合會造成在Ring Buffer填滿後，快的發送端繼續發送，接著聚合後清出Buffer空間，最後產生空洞，雖然依然可以繼續運行，但遇到這種情況時效率偏差，因此教授推薦使用Hash Table作為Buffer解決Ring Buffer會產生碎片的問題。我們實作後，發現Hash Table的碰撞太容易發生，空間的利用率比Ring Buffer還要差上許多，且由於P4的環境內不支援執行時使用迭代方式對資料結構做操作，因此在probe空的暫存器上我們並沒有想到更好的方法，並維持原本Ring Buffer的設計，希望之後若有機會可以找到更好的方法。