Multilink VPN

Cole Springate and Rebecca Newman

Git: https://github.com/rebeccasnewman/Comp112-Project1

Introduction/Background

The basic aim of the project is to create a multi-link vpn that allows for bandwidth aggregation, failover reliability, and QOS. Currently most IP traffic is built for scenarios where the client and server communicate over one link. This means a typical home user with more than one available internet connection (i.e. a home internet line, a tethered cell phone, and an open wifi hotspot) can not make use of the aggregated links for increased bandwidth or reliability. Multipath TCP is under development, but is not currently supported by most home hardware and is not in the linux main branch. Even if a user is MP-TCP capable, the servers they communicate with must also be to see any benefits.
Our solution is to have a multi-link vpn running on a client with multiple network interfaces while the server software runs in the cloud on a typical single link server with bandwidth larger than the aggregate bandwidth available to the client. Packets are forwarded using UDP and the linux TUN device is used. See figure 1 for the envisioned typical use case.

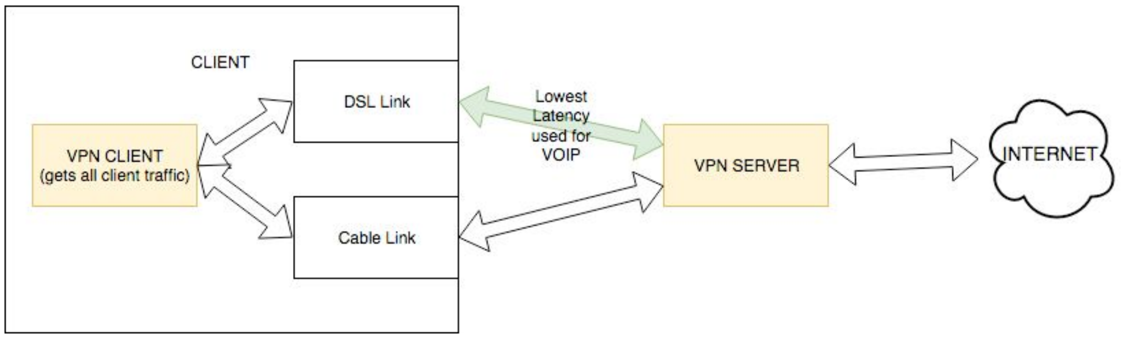
CLIENT
DSL Link
VPN CLIENT (gets all client traffic)
Cable Link
Lowest Latency used for VOIP
VPN SERVER
INTERNET

Figure 1. Diagram of the typical use case envisioned for the project

We wrote the client and server in Python 3 with one C module. The server runs on a linode VPS instance with high bandwidth, the client was tested on two laptops.

Modules / Features

Initialization
The client starts by communicating with the server via TCP to request a client id (the client will try successively from each of it's currently active interfaces). Once a client id is received, this is used to form the ip address used by the client on the vpn subnet. The client sets up a TUN interface and binds to the given ip address, and then sets a system wide default ip route with the servers vpn ip address as the default gateway. All traffic is now forwarded to the server which acts as the gateway. On the server end, packets are written out to the internet, and incoming packets are inspected for their destination ip address, which determines which client the server should send the packets to. Forwarded traffic between the client and server is encapsulated in UDP and has a custom header added.

Interface monitoring
On the client side, available interfaces are routinely polled and the links get updated to reflect changes. This means that if the client has two links, one link can be removed and new links added without dropping any ongoing connections.
In version 2, we added a TCP connection for each link. Note that all packets are still forwarded over UDP, this servers only as an indicator of link availability. This connection is initiated by the client for each link the client wants to use to communicate with the VPN. The connections are tracked by the client and server, and the client periodically sends keep alives. Since both ends monitor the connection, when the TCP connection is broken each end can react quickly and shut down that link. This allows us to take advantage of the logic of TCP to track link availability.

Server as Gateway
The server acts as a gateway for all traffic. This is accomplished by setting up some iptables rules on the server that forward traffic between the TUN interface and the server's ethernet interface. An iptables rule is also set to enable masquerading on the ethernet interface, which does NAT translation.

Link Quality for Bandwidth Aggregation
Numerous strategies were developed with the aim to aggregate bandwidth.

Packet Loss: Both the client and the server monitor link quality by calculating the rate of dropped packets on a given link. The client and server both send a UDP packet every half of a second containing information about the quality of incoming links. This is calculated by dividing the number of packets actually received by the expected number of packets. The client or server then redistribute the outgoing traffic on the links as a function of link quality.

Counting Bytes: In version two, we tried to distribute bandwidth by counting bytes. Each packet was marked as belonging to a certain bucket, and then every .3 seconds each end would tell the other end how many bytes were received in the recent bucket (ends track how many were send per bucket). Then each endpoint could updated the bandwidth metrics per link. This was done using an exponential moving average, and the link bandwidth was updated if there was packet loss or if the amount last received was greater than the current bandwidth value.
When sending packets, the endpoint would send out packets on the fastest link until it had reached the bandwidth limit, and then move on to the next link. Excess traffic was then proportionally distributed amongst amongst all link, and finally randomly distributed.

Use lowest latency only: While the counting bytes method was developed with the aim of improving upon the packet loss strategy, it didn't really do that. It was discovered that simply sending everything out the lowest latency link achieved better bandwidth aggregation. The reasons for this are discussed in the evaluation section.

Quality of Service for Speed
This module allows the client to prioritize traffic from a given IP or set of IP addresses. The quality of service is managed completely on the client side, so the server does not need to be made aware of the priority preferences of the client. The idea behind this module is to slow traffic from the low-priority addresses by setting a smaller receiver window size on the client side. This new receiver window size is scaled down as a function of priority level (1-3). Since this changes the packet, an updated tcp and ip checksum is applied with a custom c library called from python.

Quality of Service for Latency
Along with speed prioritization, a destination can be marked as low latency. The client will then only send that traffic out on the lowest latency link only, with a different action id in the header. The server will process this and then note the destination ip / client ip tuple. On incoming traffic, if the source ip and client ip match, the packet is then returned to the client on only the fasted link. Note that latency refers to the outbound latency, so it is possible that the fastest outbound link to the server from any given client will be different than the lowest latency link in the other direction. This module currently support identification based on destination ip but could be extended to identify e.g. VOIP traffic.

Quality of Service VOIP Extension

For part 2 of this project, the quality of service was extended to filter for VOIP packets and send them on the lowest latency link. This is particularly advantageous for VOIP as the quality of the call declines when packets are dropped. The challenge is that there is no way to explicitly identify VOIP packets on a link - it must be inferred from the packet information. While there are many approaches to this, we decided to use the approach defined in Matousek et. al. VOIP data is sent using IP packets with the UDP protocol. The UDP payload contains an application level header - in the case of VOIP, an RTP header. For each UDP packet that arrives at the client, we check if the source and destination ports of the packet packet are registered ports (their port number is greater than 1024). These ports are available for use by applications upon registration with IANA. If the packet meets this criteria we then check for the possibility that it is an RTP packet. We implemented both flow-based and packet-base RTP check as outlined in Matousek et. al.

Figure 2. RTP Packet Header

For packet-based checking, we first attempt to unpack the first 12 bytes of the UDP payload into the beginning of an RTP header. Note that another application header may unpack in a similar way, so there is a change of false positive classification. If the header unpacks the way we are expecting for RTP, we check that the RTP version number (first two bits of the header) is 2. The paper also suggested to check whether the padding bit is set (bit 3 of the header), but many applications and hardware may set this without properly accounting for the last byte of the data, so we chose to ignore this bit. The last per-packet test was to make sure that the payload type is set to a number in the RTP “dynamically set” range.

While the above filtering does find VOIP packets, there are many ways for false positives to happen. To lower the number of false positives, we implemented per-flow RTP checking as well. Once a packet is labeled as VOIP in the per-packet check, we check if we have seen any other packets with the same source and destination IPs and port numbers. If this tuple has already been seen, we check to make sure that at least half of the other packets in that flow were categorized as VOIP before officially classifying the current packet. This deviates from the process outlined in Matousek et. al. slightly, in that they simply set a numeric threshold (> 10 VOIP packets). They showed that this is not always so accurate.

Connection Failure Recovery

The system itself should not fail when one or more of the connections fails. Therefore, we implemented a recovery system that allows for one or more of the interfaces to fail while still serving the request. Both ends monitor the available links and if a link fails it is closed. The client also checks for new interfaces periodically, and add those when they become available.

Mobility

Although this was not an explicit aim of the project, this vpn does allow for an increased degree of mobility. Since a client can remain "active" with the server without any active links (for a period of time), it is possible for the client to lose all links and then reconnect over a new link, all while maintaining a tcp connection. For example, this allows a client to change wifi networks mid download, and have the download continue without the tcp connection breaking.

Packet Reordering

Due to the different latencies involved in using multiples links, out of order packets are more common, which can cause duplicate acks and then TCP can slow down. With the aim of alleviating this, packet reordering was implemented.

Version 1: Every outgoing packet had a monotonically increasing ID added to the header, and we reordered the packets on both ends of the TUN device. Out of order packets were buffered and sent at the appropriate time. To account for missing or dropped packets, the buffer is simply written to the TUN device if there is a delay in receiving the expected packet. This strategy involves one buffer per client, and all packets (including UDP) are reordered.

Version 2: To improve on this, we added per stream TCP reordering in version 2. This was done on the client side by tracking ACKs and SYNs, and so does not require any additional information in the packet header. This allows for a more reasonable MTU size. This strategy involved one buffer per TCP stream, and currently has one timeout value before the buffer is emptied. A future improvement could involve determining a good timeout value based on RTT or similar.

Evaluation

Bandwidth Aggregation
Bandwidth aggregation was tested on a laptop connected to two relatively low speed wifi hotspots. The results of the bandwidth gains from link aggregation using the dropped packet strategy are shown in table 1. Without packet reordering, we see that the bandwidth when aggregating is about the same as the slowest link. The results with reordering are more interesting, the speedtest tests show more bandwidth available than either link individually, but not full utilization. The TCP download is now as fast as the fastest link, but no faster. Further thoughts on this are discussed in the challenges section.

Table 1. Bandwidth aggregation results for the packet loss strategy

	speedtest download (Mb/s)	**speedtest upload (Mb/s)**	**TCP download (KB/s)**
Link One	6.21	1.86	1010
Link Two	3.35	2.31	397
Both Links - no reordering	3.93	3.14	488
Both Links - with reordering	7.99	4.20	995

Using the byte counting strategy, results are similar to those in Table 1. There was not saturation of both links and speeds were generally no faster than the fastest link. However, as mentioned earlier always using the lowest latency link allowed for slightly better results. Testing at Tufts produced the following results:
Using both links: 3.47 MB/s
Using only fastest link: 2.90 MB/s
This did not saturate both links, but it did allow for a higher bandwidth than using either link alone. This was somewhat surprising because all traffic was sent out only one link for each 0.3 seconds. We believe the better performance of this strategy is because it allowed bandwidth to be aggregated based on the routers buffers filling up. When the buffers of the routes along each link filled up, that link slowed down and the other link would become the fastest link, etc. In the future perhaps a more advanced lag based strategy would perform well. The benefits of reacting to lag is that a different link can be chosen before packet loss, which is important to keep the TCP speed high enough for saturation.

Speed QOS

To test the quality of service, we monitored two simultaneous downloads from IP addresses with varying priority. The first IP was a large download to saturate the links. We found a baseline download speed for the second IP by setting both of the IPs to have the same low priority of 3. Then, we set the second IP to have a priority of 1, and saw an increase in the average download speed. Table 2 shows the results for the download in question.

Table 2. Speed Quality of Service Results

	Average Download Speed	**Total Download Time**
Both Low Priority	179 Kb/s	1m54s
One High, One Low	328 Kb/s	62s

Low Latency Quality of Service

To test the low latency QOS, ping was run on a client with two network interfaces with the google dns server as the destination (which is always fast). The results are shown in Table 2. The minimum is the same since without QOS some of the packets will follow the fastest route, while the average with QOS is twice as fast and the maximum and deviation are much lower with QOS enabled. These results suggest that this module was successful in ensuring certain traffic always uses the fastest route.

Table 2. Round Trip Time

	Min	**Avg**	**Max**	**MDev**
No QOS	11.03	23.03	108.08	20.40
With QOS	10.53	11.74	12.80	0.47

Quality of Service VOIP Extension

To test the VOIP extension we ran our system and first used the voice chat application Discord. The system found VOIP packets while this was running. We then turned off the chat application and ran DNS lookup, which is another UDP-based application. The system did not flag these packets as VOIP.

Fail-over

Given a client with two network interfaces, we expect that a tcp connection can remain active over the vpn as the interfaces come up and down. This was tested on a laptop with built-in wifi and a usb wifi card as follows: start the vpn, start a download, physically remove the usb wifi card, wait, reinsert the card. It was tested at Tufts (on the Tufts and EECS wifi) and in a home setting (home wifi and a public hotspot). As expected, the download speed was affected by the network changes, but never disconnected.

Mobility

At tufts, a client with only one active interface was used for testing. The client was started, and a download was started, and then the wifi was switched to a different network (between eecs and tufts secure). This caused the download to briefly fall to 0, but it picked back up without being interrupted. Contrast this to switching wifi networks without using the vpn, the download will never continue as the tcp connection breaks.

Challenges

- We aimed to have the vpn allow for complete bandwidth aggregation, where the bandwidth available to the client would be the sum of the link bandwidths. We were unable to fully achieve this. There are a few factors identified that could be causing this. First of all, we allocated how much to send out on each link by using the previous packet loss as a penalty, and then we counted use based on number of packets sent. Improvements around the penalty function could be made, and we could count the bytes going out each interface rather than the packets. Before implementing reordering, the maximum TCP download speed achieved seemed to be that of the slowest link. This was very likely due to the packets arriving in a different order and trigger TCP congestion controls. Once reordering was implemented, the speed was greater than that of the slowest link, but still did not fully saturate the faster link. It is possible that either the reordering module or the link allocation logic could be improved to allow for full saturation.
- It was hard to see noticeable differences in download speeds when testing on the Tufts network because the link speed was too fast (the maximum speed through the VPN was much lower than the link speeds). We instead tested many of the modules on a home network, where average link speed is much lower.
- TCP Timeouts. Using different links with different latencies introduced a few difficulties. While the reordering module helped alleviate packets arriving out of order, it does not fix TCP timeouts. It’s possible that the RTT value would get set too low if traffic went over the fastest link for a while, and then when traffic moved to the slower link, the sender would timeout unnecessarily. It could be possible to introduce lag on faster links to try to equalize RTT amongst the links. Alternatively, the VPN could instead act more as a proxy and actually send ACKS itself, and then be responsible for getting the data to the client perhaps using something like selective ACK which is better suited for the multiple links. This last strategy might be best suited for large file transfers.
- A server based tcp packet cache was explored, but was not successful. The idea was that packet loss was most likely to occur between the server and the client, so that putting a cache on the server could hide packet loss from the sender and

keep the TCP connection from reducing speed. This is potentially still a good idea, but our version did not seem to help.

References

Matoušek P, Ryšavý O, Kmet M. Fast RTP detection and codecs classification in Internet traffic. *Journal of Digital Forensics, Security & Law* 2014; 9(2):101–112.

https://tools.ietf.org/html/rfc3551