Bandwidth Aggregation in wireless interfaces

Reports and Observations
Supratik Singha – 108997392 – **CSE524**

Introduction:

In this project we have tried to analyze the pros and cons of attempting to attain an aggregate bandwidth between two wifi interfaces on Avila board by Gateworks using XR7 cards from Ubiquiti Networks. The operating system used was OpenWRT with Linux 3.10. There are a number of different approaches we tried and we list out the challenges faced during the course of our experiments. The end of the report also talks about OpenWRT configurations and compilation instructions to make it easier for future follow up works.

Background:

Unlike Ethernet intefaces wifi interfaces face a lot in challenges in the way they transmit frames to each other in the physical layer. A wifi interface who is transmitting on some frequency range and channel may also have many other wifi interfaces transmitting in the same frequency range and channel number. Due to the fundamental property of wireless transmission that certain interfaces may be hidden from a sender, it's more difficult to detect collisions. Hence collision avoidance is used and in this protocol a wireless interface may have to defer sending frames for a random amount of time which varies widely. Hence it's not possible to attain a guaranteed bandwidth during wireless communication. What we are trying to do here is when a node has multiple interfaces (multihomed) we should be able to load balance traffic among different options depending on whether any of the interfaces is facing high or low interference or not. In other words we are trying to attain an aggregate bandwidth out of two or more wireless interfaces in a multihomed node.

Approaches:

We have mainly tried to use the following software solutions to try and experiment with bandwidth aggregation.

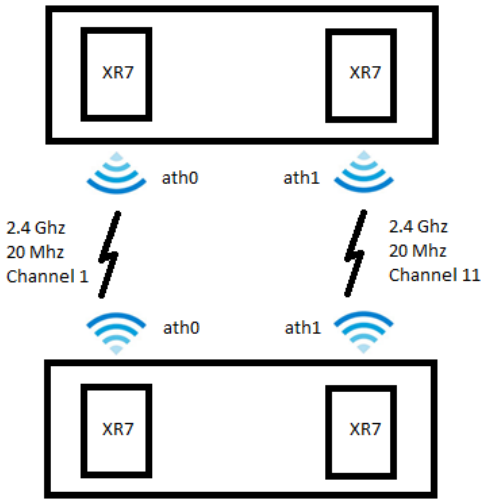
1. Bonding
2. Virtual Tunneling
3. SCTP and Multipath TCP

Pre-requirements:

We are working with XR7 cards on Avilla board by Gateworks. We are using OpenWRT Linux 3.10. The Avilla board is shipped with drivers supported only on older kernels. We needed a newer kernel to get the bonding module support.

Setup:

Two XR7 cards were connected to the each Avilla board. We made wireless interfaces on both cards ath0 and ath1 with wlanmode as ad-hoc. Such that ath0 on one end connects to ath0 on the other end. And ath1 on one end connects to ath1 on the other end. To keep the communication among the wifi interfaces as independent to each other as possible, we configured the ath0 interfaces to 2.4 Ghz, 20 Mhz and channel 1 and ath1 interfaces to 2.4 Ghz, 20 Mhz and channel 11 as shown. We also configured the the ip addresses to both the interfaces to different netmasks.



Bonding:

Bonding module comes with Linux 2.6 kernel and above and hence cannot be used with older kernels. Hence we had to bring up the wifi cards with 3.10 kernel. We tried to enslave ath0 and ath1 with default settings which is round robin distribution of packet transmission on each interface in equal amounts. We found that the resulting bandwidth gave really bad results. On sending 10 ping packets we found 50% packet loss. Now since each individual interface was more or less around the same bandwidth by themselves, it doesn't make sense to distribute packet transmission in accurately weighted amounts. We tried to find the individual behavior of the interfaces and then building up in steps.

We tried to find out if there are any issues when both the interfaces are being used simultaneously, even though thay are on separate channels. Tried pinging from both sides simultaneously on separate links and track packet loss. We used with huge packet size (65507 bytes). We found no packet loss in 120 pings.

300 to 350 ms delay on an average on ath0 link
350 to 500 ms delay on an average on ath1 link
No packet loss.

Tried iperf with UDP packets from both sides simultaneously on separate links and tracked bandwidth to look for possible dependencies among the interfaces. Each link gives reasonably lesser bandwidth than individual performance. But not drastically less than the other. About 0.2 Mbps less than individual performance.

ath0 to ath0 (with ath1 running too) [3] 0.0-360.7 sec 99.1 MBytes 2.31 Mbits/sec
ath0 to ath0 (with ath1 not running) [3] 0.0-360.4 sec 117 MBytes 2.71 Mbits/sec
ath1 to ath1 (with ath0 running too) [3] 0.0-360.5 sec 103 MBytes 2.40 Mbits/sec
ath1 to ath1 (with ath0 not running) [3] 0.0-360.5 sec 107 MBytes 2.48 Mbits/sec

This rules out any possibility that both interfaces may not be working together because of interference of channel overlap or any other reasons. iperf with TCP connections was not used because of possible retransmissions that might happen due to packet loss. Also verified the results by running scp from both sides.

We then tried to bond (ifenslaved ath0 and ath1) the interfaces and did similar experiments. Monitored the number of packets sent and received by the bond0 interface. The following counters are the sum of the corresponding counters of the individual ath0 and ath1 interface. Considering ath0 to ath0 as a separate link and ath1 to ath1 as separate link. Both are on distant channel numbers 1 (ath0) and 11 (ath1).

ifenslaved ath1 ath0:

```
ath0  <--- 2         60 <---  ath0    (58  packets lost)
      ---> 207       4 --->           (207 packets lost)
ath1  <--- 60        60 <---  ath1
      ---> 207     207 --->
```

ifenslaved ath0 ath1:

```
ath0  <--- 311   ??  307 <---  ath0
      ---> 298       301 --->
ath1  <--- 7         306 <---  ath1   (299 packets lost)
      ---> 298         6 --->         (292 packets lost)
```

We see that in the first experiment, ath1 is the master interface and ath0 is the slave interface. We see out of 60 packets transmitted via ath0 only 2 reached the receiving side, hence 96% packet loss. Similarly out of 207 packets transmitted through ath0, only 4 reached the receiving side, hence 98% packet loss. But the master interface has no packet loss. We also see that the bonding module tries to push equal number of packets on both its master and slave. But not both the interfaces are getting bonded equally. The interface mentioned first in ifenslave (master) gets bonded properly and almost 95% of the traffic succeeds RX / TX through that interface. The bonding does not try to send 95% of traffic through ath1. The bonding driver 'pushes' same amount of traffic through both the interfaces, but only ath1 is able to RX / TX the packets

and not ath0. Hence proving it's only in the lower layer at the individual interfaces due to which were are getting packets dropped.

Bonding works in a way that when it bonds multiple interfaces it tries to change the mac address of all of its slave interfaces. The slave interfaces are tried to be given the same mac addresses as the master. In our case ath1 mac address is changed to that of ath0. Now although the ath1 mac address gets changed it doesn't change the mac address of the underlying base device wifi1.

Bottleneck and Conclusion: ifconfig hw ether cannot be used to change mac address of wifi base devices like wifi0. We tried to use macchanger which did change the mac address and we were able to reach a setup wherein all the interface mac addresses were same (ath0, ath1, wifi0, wifi1 and bond0) but macchanger actually spoofs the mac address in a way that the interface does not receive packets sent to the changed or spoofed mac address. Hence macchanger does not entirely solve the problem. We may try to configure the interface in promiscuous mode to see if we are able to receive packets sent to a changed mac address.

Virtual Tunneling:

We tried this approach to remove any dependency on mac address changes in the underlying network interfaces used by bonding. vtund is a tunneling daemon that can be used to create tunnels between facing interfaces. vtund basically creates TCP connections between the interfaces and it stays there throughout the lifetime of the daemon / service. It creates interfaces on top of the tunnels. We tried to bond the tunneled interfaces and got the following results.

tap1 and tap2 are the tunneled interfaces created on top of ath0 and ath1 respectively. We tried similar experiments as in case on normal bonding by sending huge data from one side or both sides and capturing the packet counts on the individual interfaces. (In this case we could not use iperf because iperf reports heavily erroneous results in its bandwidth. For example even though it reports the right bandwidth of 3.5 Mbps on the tap2 interface, when bonded with tap1 interface, it produces 45Mbps bandwidth on the bond0 interface. Please refer to attached raw results: iperf_bond0_tunnel.txt). Hence we created a utility that sends packets from a UDP client to UDP server which measures average and current bandwidth.

UDP Bandwidth tool utility: https://github.com/susingha/wifiAggregationTools

ifenslaved tap1 tap2: vtunnel over ath0 and ath1

bond0	<--- 34829	34829 <---	bond0	
	---> 31690	31690 --->		
ath0	<--- 34431	34432 <---	ath0	(1 packet lost)
	---> 32954	32861 --->		(93 packets lost)
ath1	<--- 31848	32324 <---	ath1	(476 packets lost)
	---> 32771	32508 --->		(263 packets lost)
tap1	<--- 17414	17414 <---	tap1	
	---> 15845	15845 --->		
tap2	<--- 17415	17415 <---	tap2	
	---> 15845	15845 --->		

What we find here is that there has been relatively less packet loss percentage than what we saw in case of pure bonding and that the total number of packets pushed through the bonded interface has actually been transferred to the other end. But we also see that the number of packets pushed through the tap1 interface is not same as the number of packets pushed through the ath0 (underlying interface of tap1), in fact much more, implying that there has been reasonable amounts of packet drops and retransmissions between the TCP tunnel.

Let up check the UDP bandwidth measurements:

tap1 to tap1:

22790144 Bytes. Average bw = 455802.88 Bytes/s, Current bw = 459571.19 Bytes/s

tap2 to tap2:

3973120 Bytes. Average bw = 397312.00 Bytes/s, Current bw = 425164.81 Bytes/s

bond0 to bond0: try1:

1200128 Bytes. Average bw = 171446.86 Bytes/s, Current bw = 240025.59 Bytes/s
1204224 Bytes. Average bw = 60211.20 Bytes/s, Current bw = 819.20 Bytes/s
1314816 Bytes. Average bw = 52592.64 Bytes/s, Current bw = 22118.40 Bytes/s

bond0 to bond0: try2: (vtund server cannot be made UDP server)

143360 Bytes. Average bw = 15928.89 Bytes/s, Current bw = 28672.00 Bytes/s

We see that the aggregate bandwidth measurements (on bond0) does not add up to the individual bandwidths of tap1 and tap2 but rather fluctuates heavily in a huge range.

Bottlenecks and Conclusion: We may go forward with more experiments with tunneling but we found that particularly vtunnel is too buggy to be scalable and crashed our system at times. Also since it depends on a TCP tunnel as the underlying protocol, it may raise additional limitations on what protocols we may run on the tunneled interface.

SCTP:

We mainly referred the paper on bandwidth aggregation over SCTP. SCTP is already a protocol which supports multipath streaming of connection oriented data flows. It is also able to switch among interfaces in a multihomed host. Hence implementing an SCTP approach towards bandwidth aggregation reduces to the way in which we distribute traffic flows among participating interfaces. Although this requires background enhancements like implementation of host level association between endpoints, we started off by writing a small SCTP communication tool using SCTP sockets between a client and a server.

SCTP Client Server tool: https://github.com/susingha/wifiAggregationTools

Multipath TCP:

We could not go much forward with this protocol because integrating the MTCP enabled kernel source code with OpenWRT was a major bottleneck. In what we did have the MTCP up in a normal PC with GNU Linux but the given hardware did not support bringing up the wireless cards (XR7) using PCI cards.

OpenWRT - Installation, Compilation, Flashing – Overview:

Ref: https://github.com/susingha/wifiAggregationTools/blob/master/OpenWRTSetupInstructions.txt

Challenges and future work:

1. Board bringup with Linux 3.10, flashing image from RedBoot.
2. Getting drivers for XR7 card. Tried with ath5k, ath9k and madwifi: The XR7 card came up only with the madwifi driver. The madwifi driver given at the Gateworks website work only for the Linux 2.4 kernel and not with Linux 3.10 kernel. The madwifi driver included in the Linux 3.10 kernel needs required patches for detecting the XR7 cards to appear.
3. Madwifi driver did not support changing of channel width from 20 Mhz to 40 Mhz. The proc interface provided with the Madwifi driver only reports the current bandwidth setting being used but does not allow it to be changed. The driver code that deals with any writes on the proc file leads to assembly code.
4. Changing mac addresses on wireless interfaces: wlanconfig uses the base device wifi0 to create wireless extensions like ath0. The mac addresses on the wireless extension needs to be same as the base device. Although

we can change the ath0 mac address using ifconfig hw ether, we cannot change the base device mac address in the same way. macchanger was used to change the base device mac address but this utility can only spoof the the changed mac address on the network. Changing the mac address does not really result in receiving packets that are intended for the spoofed mac address.

5. Multipath TCP is implemented in its own kernel code. Integrating this kernel with the OpenWRT code base is complicated and can be done in the long run. We tried to install Multipath TCP on a normal PC with normal kernel compilation.
6. Bringing up the XR7 card on a normal PC with Linux 2.4 kernel as well as Linux 2.6 kernel was tried with PCI slots. The cards could not be brought up.
7. SCTP needs enhancements like implementation of host level association between endpoints and other core enhancements at the transport layer to support multihoming in addition to just using multihomed interfaces as a back up interface in case of failure.
8. MTCP and SCTP needs end to end protocol support.

Readings and Raw Data:

All readings and raw data observations (self explainatory) are available at:

https://github.com/susingha/wifiAggregationTools/tree/master/Project%20WIFI%20Readings