TCP Over SoNIC

Xuke Fang
Cornell University

XingXiang Lao
Cornell University

ABSTRACT

SoNIC [1], a Software-defined Network Interface Card, which provides the access to the physical layer and data link layers in software by implementing them in software, provides complete control over network in real time; Hence it gives system programmers unprecedented precision for network measurements and researches. However, SoNIC only supports UDP transmission protocol for now. In this paper, we present the implementation of TCP over SoNIC and evaluation of TCP throughput over different configurations.

1. INTRODUCTION

Physical layer(PHY) is often implemented in the hardware and is neither accessible to systems programer nor to operating systems. Hence it is often viewed as a black box for systems programmer. However, with the network growing faster, the optimization on the top network layers hits a bottleneck, and increasing number of researchers has started to focus on the optimizing PHY. There are few tools available for accessing PHY, like BiFocals [2] and SoNIC. BiFocals uses physical equipment, including a laser and an oscilloscope. SoNIC, on the other hand, is another powerful tool that gives systems programmers and researchers the access to the PHY and data link layer by implementing the PHY in the software. SoNIC is able to generate packets at full data rate with minimal inter-packet delay. It also provides fine control over the incoming packets with sub-nanosecond granularity. With the access to PHY from software, SoNIC gives researchers untapped potential to develop new applications which are not feasible before. For instance, SoNIC can precisely measure the available bandwidth by measuring the number of idle packets between two packets, increase the TCP throughput and characterize network traffic.

Unfortunately, with all the benefits that SoNIC could provide, before this paper, SoNIC is only optimized to generate UDP packets. In other words, the limitation to UDP packets narrows the application of SoNIC to support only unreliable transmissions. As the result, we implemented TCP network stack on top of SoNIC in order to fully extend the power of SoNIC. Implementing TCP over SoNIC is quite challenging because TCP has to maintain a state machine while still satisfying real time constraints at line rate. In this paper, we will present our implementation of TCP over SoNIC in detail and its throughput in different tests.

2. BACKGROUND

The FPGA board that SoNIC is using is equipped with two physical 10 GbE ports. Above the PHY layer is the Media Access Control (MAC) sublayer of the data link layer. MAC communicates with the PCS sublayer of the PHY layer with a queue style data structure called FIFO. On the TX path MAC puts the information of the packets it wants to send in order into FIFO and PCS fetch and send them. On the RX path, it is the opposite direction. The same as the current UDP implementation over SoNIC, our TCP implementation will also be in the MAC layer.

Generating TCP packets is more challenging than generating UDP packets because it requires maintaining a state machine. Also to ensure that no packet loss and packet disorder will occur packet resend can not be avoided. Resending packets is a main factor that affects the data transmission speed. To reach a high data transmission speed, a feature called âĂIJwindowâĂİ must be implemented in an elegant way. How we implemented the window feature will be mentioned later.

Proceedings of the 2011 Midstates Conference on Undergraduate Research in Computer Science and Mathematics

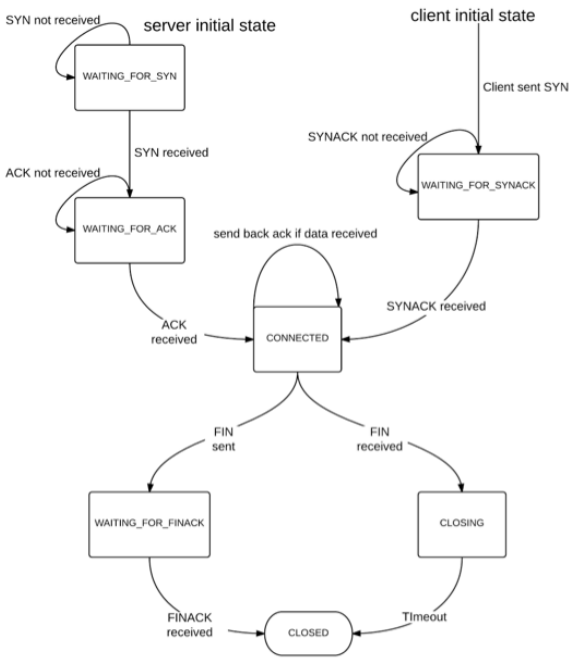

Figure 1: State Machine for the TCP implementation

3. DESIGN

3.1 TCP state machine:

As mentioned above, sending tcp packets requires maintaining a state machine. [Figure 1] is a graph of our TCP state machine.The initial state of server is WAITING_FOR_SYN and client will turn into its initial state WAITING_FOR_SYNACK after it sends the first SYN packet. There will be no difference between a server and a client after both of them turn into state CONNECTED. In state WAITING_FOR_SYNACK, a SYN packet will be resent if SYNACK is not received in time. The resending schemes of SYNACK, FIN and data packets are similar.

3.2 TCP data transmission and window size:

To reach a high TCP transmission speed we applied the window feature to our TCP design. To make sure every packet gets delivered to the destination, TCP requires the receiver to reply the sender with an ACK packet every time it receives a data packet. And the sender cannot send the next data packet before it sees the ACK packet corresponding to the last data packet it sent. This enforces the sender to send one packet at a time, which is very slow. To improve the speed of transmission we added the âĂIJwindowâĂİ feature to our TCP design. Window is the number of packets that can be sent at a time. In other words, the sender is allowed to send window size number of data packets and before it starts to listen to the corresponding ACK packets. The window size is not required to be fixed. In our implementation, the sender sends idle packets for 0.3 milliseconds after it finishes sending a window number of packets. Then it checks whether it has received the ACK packet corresponding to the last packet it sent. If it has received it, then it means all data packets sent previously have been received. It will increase the window size by a constant and start to send the next window size number of packets. If the sender didn not receive the ack it was supposed to received, then it means we may have lost some packets during the transmission or the transmission speed, the window size, is too big for the current network environment to handle. Then It decreases the window size by half.

4. IMPLEMENTATION

4.1 Filling packets contents and unmarshalling the packet

For each packet, we fill in the Ethernet header first and fill in the ip header in the ethernet frame payload. And at last we fill in the TCP header in the ip packet payload and fill in the TCP payload.

To unmarshall the packet we unmarshall Ethernet header, ip header, tcp header in order.

4.2 Sending and receiving packets

Thread MAC_TX puts packets into the FIFO for PCS_TX to fetch and send. PCS_RX receives packets and forwards them for MAC_RX to process. Figure 2 shows how port 0 sends a data packet to port1 through MAC_TX0, PCS_TX0, PCS_RX1, MAC_RX1 and port 1 replies port 0 with a response packet through MAC_TX1, PCS_TX1, PCS_RX0, MAC_RX0. .

4.3 TCP state machine

The receive thread(sonic_mac_rx_loop) checks the flag of the packet received and the current state to decide whether the packet needs a response and whether the state needs to be changed.

The send thread(sonic_mac_pkt_generator_loop) sends packets corresponding to the current state. We added a timeout after each packet or window size of packets have been sent. After the timeout we check the current state to see whether we need to resend to the packet or to send next the packet indicated by the state machine described above.

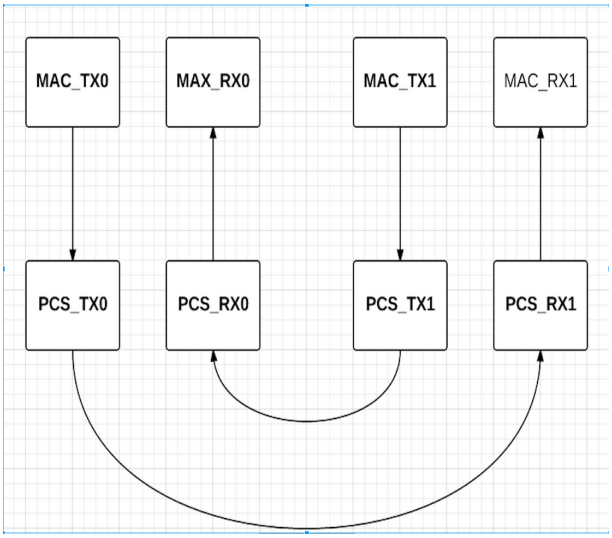

Figure 2: Schematic of SoNIC Implementation

4.4 Timeouts

The physical layer of SoNIC requires MAC_TX to keep feeding it with packets to send, which means we cannot use blocking calls like sleep to act as a timeout timer. Instead, we are using sonic_gen_idles() to generate idle packets that take a certain time to be sent by the physical layer. For example, after the client sends the first SYN packet we call sonic_gen_idles() to generate idle packets that will be sent in 0.1 milliseconds and check whether we received the SYNACK response after the generated idles packets have been sent.

4.5 Seq, Ack and Window

The sender keeps the current seq number and receiver keeps the current ack number. When the sender receives an ACK packet it sets the current sequence number to be the ack number of the ACK packet if it is larger than the current seq number. When receiver receives a data packet it checks whether the sequence number is the same as its current ack number. If yes, then update the ack number, else, do nothing because in this case this data packet is either a duplicate or we have not received all data packets before it.

To implement the window feature, now the sender sends a window size of packets starting from the packet with the current seq number but it only checks whether the ACK packet corresponding to the last packet sent in the last window has been received or not and adjust the window size.

4.6 What we have not done

For now the algorithm used to change window size is very simple. Our window size always increases linearly and decreases by half.

5. EVALUATION

In this section, we present the throughput of the TCP implementation on top of SoNIC with a loopback configuration. Here, the loopback configuration means that TCP connection is established between two ports of same SoNIC board. In addition, we also present the technique that we used to improve the maximum throughput of TCP connections.

In order to improve the performance(throughput) of TCP connection, understanding of the limiting factors of the TCP throughput is necessary. One trivial limitation is the maximum bandwidth of the slowest link in the path. However, since the TCP is sending and receiving the packets in a loopback configuration and the bandwidth between these two ports are much larger than the throughput. Hence, the bandwidth is not the bottleneck of the throughput. The equation below shows the maximum throughput is bounded by the maximum segment size (MSS), round trip time(RTT) and possibility of packet loss (Ploss).

$$Througput \leq \frac{MSS}{RTT \cdot \sqrt{P_{loss}}} \tag{1}$$

However, since the TCP is running in loopback configuration and the packet loss is so rare that the TCP window becomes regularly fully extended, this formula also doesn't apply. This leads us to the following equation:

$$Througput \leq \frac{RWIN}{RTT} \tag{2}$$

Where RWIN represents the TCP receive window size and RTT represents the round trip time. Even with no packet loss, the receive window size still has a significant impact on the TCP throughput. This is because that sender will keep sending data to the receiver until the it reaches the window size before waiting for the acknowledgement from the receiver. Then if the sender does not receive the acknowledgement from the receiver for a certain period of time, it retransmits the data to the receiver. This will cause a significantly decrease on the TCP throughput. The round trip time is another factor that can significantly impact the throughput as

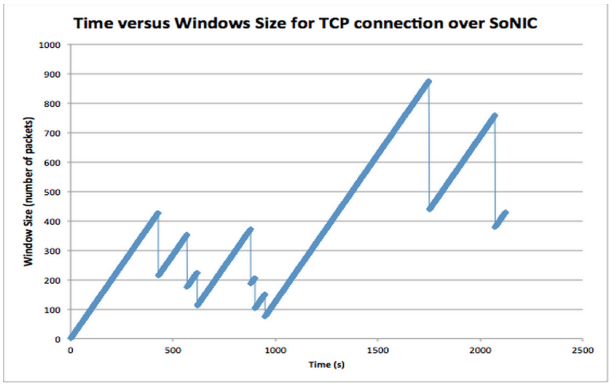

Figure 3: Time versus Receive Window Size [retransmission timeout = 0.2ms; 100000 pkts]

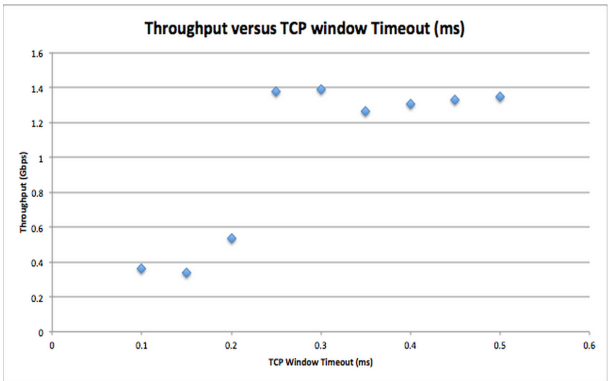

Figure 4: Throughput versus Retransmission timeout [100000 pkts]

shown in the equation. However, we are not able to tune the RTT since it is a physical limitation imposed by the wire length, available bandwidth and etc. But we are able to tune the retransmission timeout to generate the maximum throughput.

Firstly, the receive window size keeps increasing linearly if sender can receive the highest acknowledge number of the packets that were sent in time. If sender does not receive the highest acknowledge number before the retransmission timeouts, sender will retransmit data to the receiver and decrease the receive window size. Figure 3 shows the behaviour of the window size. As mentioned before, retransmission will significantly impact the throughput. Hence ideally, window size should keep increase and never drop. The main reason causes

window size to drop is that the retransmission timeout is too short and the sender is not able to receive the corresponding acknowledgement in time. Therefore, we try to increase the retransmission timeout and plot versus throughput, which is shown in the Figure 4. As shown in the figure, the throughput initially increases as the retransmission timeout increases before it hits a plateau, then the throughput becomes around 1.4 Gbps. The initial increase of throughput is caused by decreasing the number of data retransmission. When the retransmission timeout is long enough that there is no retransmission needed, the throughput flattens. Theoretically, throughput should decrease as the retransmission timeout keeps increasing after it hits the highest point. However, since the retransmission timeout is so small comparing to time to transfer the actual data, the increase on retransmission timeout has little impacts on the overall throughput. As mentioned before, the configuration is loopback; hence the characteristic of the network traffic and topology is predetermined. Therefore, the window size can be preconfigured to the maximum window size that the network can handle before running the test and then run the test with this maximum window size throughout the test. This should yield the maximum throughput. However, the result throughput is approximately equal to the throughput that was obtained when increase the window size linearly. Hence, we conclude that the bottleneck of the throughput is not the window size and uture research is needed to improve the throughput.

6. CONCLUSION

In this paper, we presented the TCP implementation over SoNIC and the techniques we used to improve the maximum throughput. The TCP implementation is robust and is able to transmit data between two SoNIC ports reliably. Also we were able to achieve the maximum throughput of TCP connection to be around 1.3 Gbps. This is still lower than the theoretical maximum throughput. Hence, future research is needed to achieve the theoretical maximum throughput.

7. FUTURE WORK

For the future work, we need to improve performance of the TCP throughput to ideally 10 Gbps . Also we have to measure performance of TCP with different characteristics

8. ACKNOWLEDGMENT

We are grateful to the following people for resources, discussions and suggestions: Prof. Hakim Weatherproof, Ki Suh Lee.

9. REFERENCE

[1]. Lee, Ki Suh, Han Wang, and Hakim Weatherspoon. "SoNIC: Precise Realtime Software Access and Control of Wired Networks." 1 Jan. 2013. Web. 13 Dec. 2014. <https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final138.pdf>.

[2]. D. A. Freedman, T. Marian, J. H. Lee, K. Birman, H. Weather- spoon, and C. Xu. Exact temporal characterization of 10 Gbps optical wide-area network. In Proceedings of the 10th ACM SIG- COMM conference on Internet measurement, 2010.