Project Report

TCP over SoNIC

Group Members

Name	Email ID
Abhishek Kumar Maurya	am2633@cornell.edu
Amarinder Singh Bindra	ab2546@cornell.edu
Gaurav Aggarwal	ga286@cornell.edu

Table of Contents

1. Project Summary

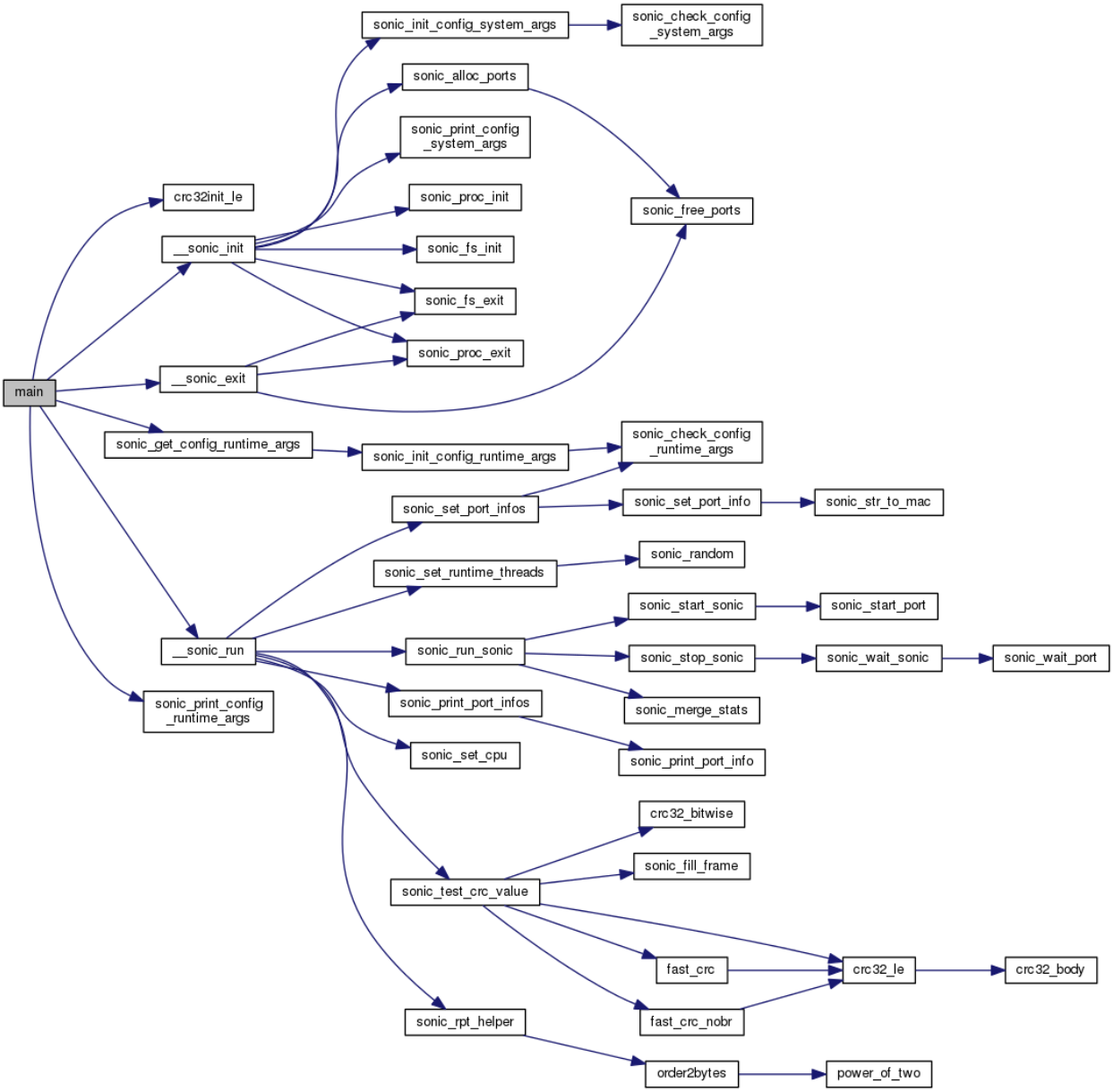
This project involves implementation of Transmission Control Protocol (TCP) layer on top of Software-defined Network Interface Card (SoNIC). SoNIC currently is optimized to generate UDP packets because of real-time constraint. Generating TCP traffic over SoNIC requires maintaining a TCP state machine for tracking connection state and reliable transfer of data while satisfying real-time constraints and line speed. We target to build a TCP layer on top of SoNIC that operates at line speed.

2. Implementation Stages

Implementation Stages	**Status**
Understand SoNIC Architecture and run UDP over SoNIC	Done
Implement TCP over UDP in User Space (Reliable Data Transfer)	Done
Implement separate TX and RX Threads	Done
Implement Cumulative ACK and Congestion Control	Done
Port to SoNIC user space	Done
Port to SoNIC kernel space	Done

2.1. SoNIC Architecture

We studied SoNIC related papers and went through SoNIC code in order to understand its architecture. We studied the already existing implementation of UDP over SoNIC. A broad level call graph of UDP over SoNIC implementation (generated via doxygen) is shown below:



We ran already existing UDP implementation over SoNIC and tried tuning its various parameters to understand their impact. We started with basic rxtx_idle mode to perform basic hardware validation by sending and receiving idle packets. Later, we ran pkt_gencap mode in order to develop deeper understanding. While running in packet_gencap mode, TX side invokes sonic_pcs_tx_loop thread which drains packets from sonic_app_cap_loop thread and puts checksum using sonic_mac_pkt_generator_loop and RX side invokes sonic_pcs_rx_loop thread which drains packets and supplies them to app.

As per our understanding so far, in order to implement TCP over SoNIC we need to implement a mode similar to pkt_gencap which in addition to its current capabilities, maintains a TCP state machine to track connection state and uses acknowledgements and re-transmission to achieve reliable packet transmission.

Following are the logs which we captured while running existing UDP implementation over SoNIC:

```
cs5413_netbusters@compute30:~/sonic/driver/kernel$ echo tester mode=pkt_gencap,pkt_gencap pkt_len=1518
duration=5 idle=13738 pkt_cnt=10000 wait=3 mac_src="00:60:dd:45:39:0d" mac_dst="00:60:dd:45:39:5a"
ip_src="192.168.4.12" ip_dst="192.168.4.13" port_src=5000 port_dst=5008 vlan_id=2200 gen_mode=0 >
/proc/sonic_tester

Oct 28 23:31:33 compute30 kernel: [ 5539.660616] SONIC: [sonic_run_sonic]
Oct 28 23:31:33 compute30 kernel: [ 5539.660621] SONIC: [sonic_start_sonic]
Oct 28 23:31:33 compute30 kernel: [ 5539.660983] SONIC: [sonic_run_thread] Thread rx_pcs0 is running on 10
Oct 28 23:31:33 compute30 kernel: [ 5539.661002] SONIC: [sonic_run_thread] Thread rx_pcs1 is running on 1
Oct 28 23:31:33 compute30 kernel: [ 5539.661015] SONIC: [sonic_run_thread] Thread tx_pcs1 is running on 4
Oct 28 23:31:33 compute30 kernel: [ 5539.661032] SONIC: [sonic_run_thread] Thread app1 is running on 3
Oct 28 23:31:33 compute30 kernel: [ 5539.661044] SONIC: [sonic_run_thread] Thread tx_mac1 is running on 5
Oct 28 23:31:33 compute30 kernel: [ 5539.661056] SONIC: [sonic_run_thread] Thread app0 is running on 7
Oct 28 23:31:33 compute30 kernel: [ 5539.661069] SONIC: [sonic_run_thread] Thread tx_pcs0 is running on 8
Oct 28 23:31:33 compute30 kernel: [ 5539.661083] SONIC: [sonic_run_thread] Thread tx_mac0 is running on 9
Oct 28 23:31:33 compute30 kernel: [ 5539.661094] SONIC: [sonic_mac_pkt_generator_loop]
Oct 28 23:31:33 compute30 kernel: [ 5539.661101] SONIC: [sonic_mac_pkt_generator_loop]
Oct 28 23:31:33 compute30 kernel: [ 5540.193320] SONIC: [sonic_dma_tx] [p0] ??
Oct 28 23:31:38 compute30 kernel: [ 5544.658882] SONIC: [sonic_stop_sonic]
Oct 28 23:31:38 compute30 kernel: [ 5544.658884] SONIC: [sonic_wait_sonic]

cs5413_netbusters@compute30:~/sonic/driver/kernel$ Oct 28 23:31:38 compute30 kernel: [ 5544.659146] SONIC:
[sonic_run_sonic] DONE
Oct 28 23:31:38 compute30 kernel: [ 5544.659148] SONIC: [sonic_print_port_stat] ---Port [0] stats---
Oct 28 23:31:38 compute30 kernel: [ 5544.659150] SONIC: [sonic_print_pcs_stat] [tx_pcs0] time= 4999321111
pkts= 10000 blks= 732888467 err_states= 0 err_blks= 0 fifo= 0 dma= 92349
Oct 28 23:31:38 compute30 kernel: [ 5544.659152] SONIC: [sonic_print_mac_stat] [tx_mac0] time= 4996980683
pkts= 10000 err_crc= 0 err_len= 0 fifo= 0
Oct 28 23:31:38 compute30 kernel: [ 5544.659154] SONIC: [sonic_print_pcs_stat] [rx_pcs0] time= 4999300677
pkts= 302691 blks= 745630848 err_states= 0 err_blks= 0 fifo= 0 dma= 187912
Oct 28 23:31:38 compute30 kernel: [ 5544.659155] SONIC: [sonic_print_mac_stat] [rx_mac0] time= 0 pkts= 0
err_crc= 0 err_len= 0 fifo= 0
Oct 28 23:31:38 compute30 kernel: [ 5544.659157] SONIC: [sonic_print_app_stat] [app0] time= 4999281076 pkts=
302689 bytes= 459481902 fifo= 0
Oct 28 23:31:38 compute30 kernel: [ 5544.659158] SONIC: [sonic_print_port_stat] ---Port [1] stats---
Oct 28 23:31:38 compute30 kernel: [ 5544.659160] SONIC: [sonic_print_pcs_stat] [tx_pcs1] time= 4999368867
pkts= 302662 blks= 733727328 err_states= 0 err_blks= 0 fifo= 0 dma= 92455
Oct 28 23:31:38 compute30 kernel: [ 5544.659162] SONIC: [sonic_print_mac_stat] [tx_mac1] time= 4996971281
pkts= 302744 err_crc= 0 err_len= 0 fifo= 0
Oct 28 23:31:38 compute30 kernel: [ 5544.659163] SONIC: [sonic_print_pcs_stat] [rx_pcs1] time= 4999386322
pkts= 10000 blks= 743619072 err_states= 0 err_blks= 0 fifo= 0 dma= 187405
Oct 28 23:31:38 compute30 kernel: [ 5544.659165] SONIC: [sonic_print_mac_stat] [rx_mac1] time= 0 pkts= 0
err_crc= 0 err_len= 0 fifo= 0
Oct 28 23:31:38 compute30 kernel: [ 5544.659167] SONIC: [sonic_print_app_stat] [app1] time= 4999305676 pkts=
10000 bytes= 15180000 fifo= 0
```

2.2. TCP over UDP in User Space

We have implemented a TCP layer over UDP in order to provide reliable communication channel. We have implemented a TCP state machine to track connection state. We are able to

successfully establish a connection using three-way handshake and are able to reliably transfer data using acknowledgements and re-transmission.

We have written almost similar interface as actual TCP sockets provide so that the implementation is transparent to application layer. We provide description of our various components in the following sections:

2.2.1 TCP Socket

TCP Socket call creates a socket and returns a descriptor which is opaque to user and is used in later calls.

Code Snippet:

```
int tcp_socket()
{
	int i, free_fd, free_fd_found = 0;

	for( i = 0; i < MAX_SOCK_FDS && !free_fd_found; ++i )
	{
		if( socket_fds[next_fd].in_use == 0 ) //this socket fd is free
		{
			free_fd = next_fd;
			free_fd_found = 1;
		}

		next_fd++;
		if( next_fd == MAX_SOCK_FDS )		next_fd = 0;
	}

	if( free_fd_found )
	{
		tcp_socket_t *new_socket = malloc( sizeof(tcp_socket_t) );

		//initialize_created_socket
		if ( __init_socket( new_socket ) < 0 )
		{
			free( new_socket );
			return -1;
		}

		socket_fds[free_fd].in_use = 1;
		socket_fds[free_fd].socket = new_socket;
	}
	else
	{
		return -1; //error - Could not allocate fd
	}

	return free_fd;
}
```

2.2.2 TCP Connect

TCP connect call is used by a client in order to establish an active connection with the server. Calling this results in the initiation of a three-way handshake.

Code Snippet:

```
int tcp_connect( int tcp_socket_fd, struct sockaddr *addr, socklen_t addrlen )
{
        char first_packet[TCP_PACKET_SIZE], sent_ack_packet[TCP_PACKET_SIZE];

        tcp_socket_t* socket = __verify_sock_fd( tcp_socket_fd );
        if( socket == NULL )  return -1;

        //populate destination information in socket
        memcpy( &socket->conn.destination, addr, addrlen );

        //prepare and send SYN packet and receive its ACK
        __process_send_packet( socket, TCP_FLAG_SYN, first_packet );
        if( __send_reliable( socket, first_packet, 0 ) == -1 )
        {
                return -1;
        }

        if( socket->state == CLOSED ) //incorrect packet or incorrect ack number
        {
                return -1;
        }

        //send ACK
        __process_send_packet( socket, TCP_FLAG_ACK, sent_ack_packet );
        if( sendto(  socket->udp_socket_fd, sent_ack_packet, sizeof(tcp_hdr_t), 0,
                (struct sockaddr*)(&socket->conn.destination),
                sizeof(socket->conn.destination) ) == -1 )
        {
                return -1;
        }

        /* return success */
        return 0;
}
```

2.2.3 TCP Bind

TCP bind assigns the address specified by *addr* to the tcp socket created earlier using tcp_socket.

Code Snippet:

```
int tcp_bind(int tcp_socket_fd, const struct sockaddr *addr, socklen_t addrlen)
{
        tcp_socket_t* socket = __verify_sock_fd( tcp_socket_fd );

        if( socket == NULL )  return -1;

        return bind( socket->udp_socket_fd, addr, addrlen );
```

```
}
```

2.2.4 TCP Listen

TCP listen marks the socket referred to by *tcp_socket_fd* as a passive socket that is, as a socket that will be used to accept incoming connection requests.

Code Snippet:

```
int tcp_listen( int tcp_socket_fd, int tcp_backlog )
{
	tcp_socket_t* socket = __verify_sock_fd( tcp_socket_fd );

	if( socket == NULL )  return -1;

	socket->state = LISTEN;
	//TODO - tcp_backlog is ignored currently
	return 0; //success
}
```

2.2.5 TCP Accept

TCP accept accepts a SYN packet from client and sends a SYN ACK packet in response to establish a connection.

Code Snippet:

```
int tcp_accept( int tcp_socket_fd, struct sockaddr *addr, socklen_t *addrlen )
{
	ssize_t num_bytes_read = 0;
	struct sockaddr client_addr;
	socklen_t client_addrlen = sizeof( client_addr );
	char first_packet[ TCP_PACKET_SIZE ], sent_synack_packet[ TCP_PACKET_SIZE ];

	tcp_socket_t* socket = __verify_sock_fd( tcp_socket_fd );
	if( socket == NULL )  return -1;

	num_bytes_read = recvfrom(	socket->udp_socket_fd, first_packet,
					sizeof(first_packet), 0, &client_addr,
					&client_addrlen  );

	if( num_bytes_read == -1 || num_bytes_read < sizeof(tcp_hdr_t) )
	{
		return -1;
	}

	//create a new socket for communication with client
	int client_tcp_socket_fd = tcp_socket();
	tcp_socket_t* client_socket = __verify_sock_fd( client_tcp_socket_fd );
	if( client_socket == NULL )	return -1;

	client_socket->state = LISTEN;
	memcpy( &client_socket->conn.destination, &client_addr, client_addrlen );
```

```
	//we must have got a SYN packet here
	__process_receive_packet( client_socket, first_packet );
	if( client_socket->state == CLOSED )
	{
		return -1;
	}

	//send a syn_ack packet
	__process_send_packet( client_socket, TCP_FLAG_SYN | TCP_FLAG_ACK,
				sent_synack_packet );
	if( __send_reliable(client_socket, sent_synack_packet, 0) == -1 )
	{
		return -1;
	}

	if( client_socket->state == CLOSED )
	{
		return -1;
	}

	//return new socket fd -- both client and server now can
	//communicate on this socket
	return client_tcp_socket_fd;
}
```

2.2.6 TCP Send

TCP Send reliably sends the data.

Code Snippet:

```
ssize_t tcp_send(int tcp_socket_fd, const void *buf, size_t len, int flags)
{
	tcp_socket_t* socket = __verify_sock_fd( tcp_socket_fd );
	if( socket == NULL )  return -1;

	char send_packet[TCP_PACKET_SIZE];
	ssize_t sent_bytes = 0;

	//allowed to send a packet?
	if( socket->snd_nxt < socket->snd_una + socket->snd_wnd )
	{
	  __process_send_packet( socket, TCP_FLAG_ACK, send_packet );

		//__insert_send_queue(socket, send_packet, strlen(buf));

		if ((sent_bytes = __send_reliable( socket, send_packet, 0)) == -1 ) {
			return -1;
		}
	}
	else
	{
		return -1;
```

```
		}

		return sent_bytes;
}
```

2.2.7 TCP Receive

TCP receive reads the data from *tcp_sockect_fd*.

Code Snippet:

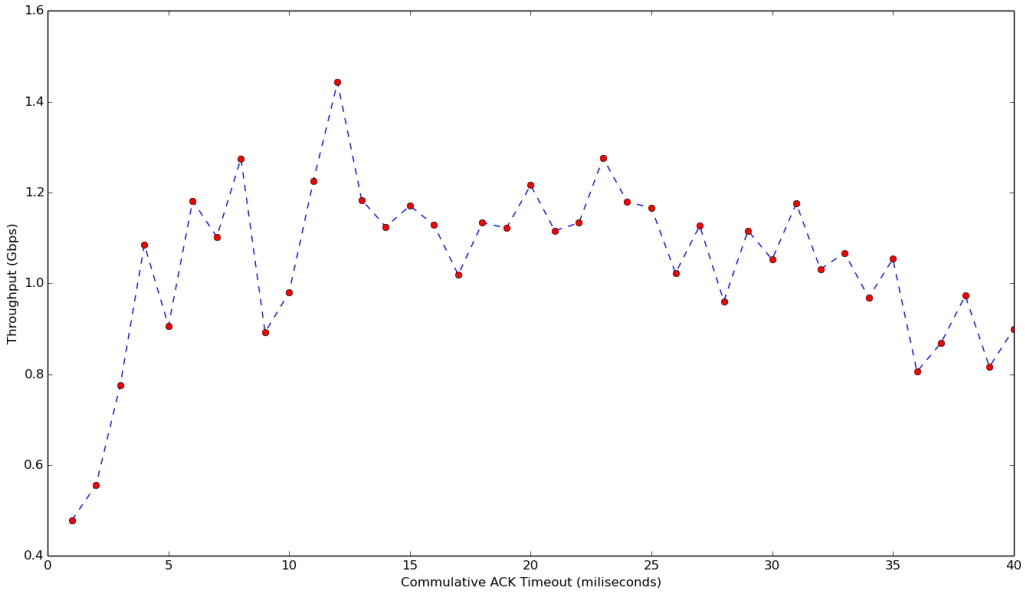
```
ssize_t tcp_recv(int tcp_socket_fd, char *buf, size_t len, int flags ) {

		tcp_socket_t* socket = __verify_sock_fd(tcp_socket_fd);
		if( socket == NULL )  return -1;

		char recd_packet[TCP_PACKET_SIZE], sent_ack_packet[TCP_PACKET_SIZE];
		ssize_t num_bytes_read = 0, num_bytes_sent = 0;
		socklen_t src_addrlen = sizeof(socket->conn.source);

		num_bytes_read = recvfrom(	socket->udp_socket_fd, recd_packet,
						sizeof(tcp_hdr_t) + len, flags,
						(struct sockaddr*)&(socket->conn.source),
						&src_addrlen);

		//update state mcahine
		__process_receive_packet( socket, recd_packet );

		//send ACK
		__process_send_packet( socket, TCP_FLAG_ACK, sent_ack_packet );
		num_bytes_sent = sendto(	socket->udp_socket_fd, sent_ack_packet,
						sizeof(tcp_hdr_t), 0,
						(struct sockaddr*) &(socket->conn.destination),
						sizeof(socket->conn.destination) );

		//ACK sent successfully?
		if( num_bytes_sent == -1 || num_bytes_sent != sizeof(tcp_hdr_t) )
		{
			return -1;
		}

		return num_bytes_read;
}
```

2.3. Separate TX and RX Threads

We have implemented separate TX and RX threads for both the sender and the receiver. The two threads synchronize with each other in order to correctly update the TCP state machine and in order to make sure that congestion window size is honored while sending data. This two threaded implementation is in line with current SoNIC implementation which has separate TX and RX threads and therefore is easier to port to SoNIC.

2.4. Cumulative ACK and Congestion Control

Additionally, we have implemented cumulative ACK in anticipation of improving performance and we get following results as we vary the cumulative ACK timeout:

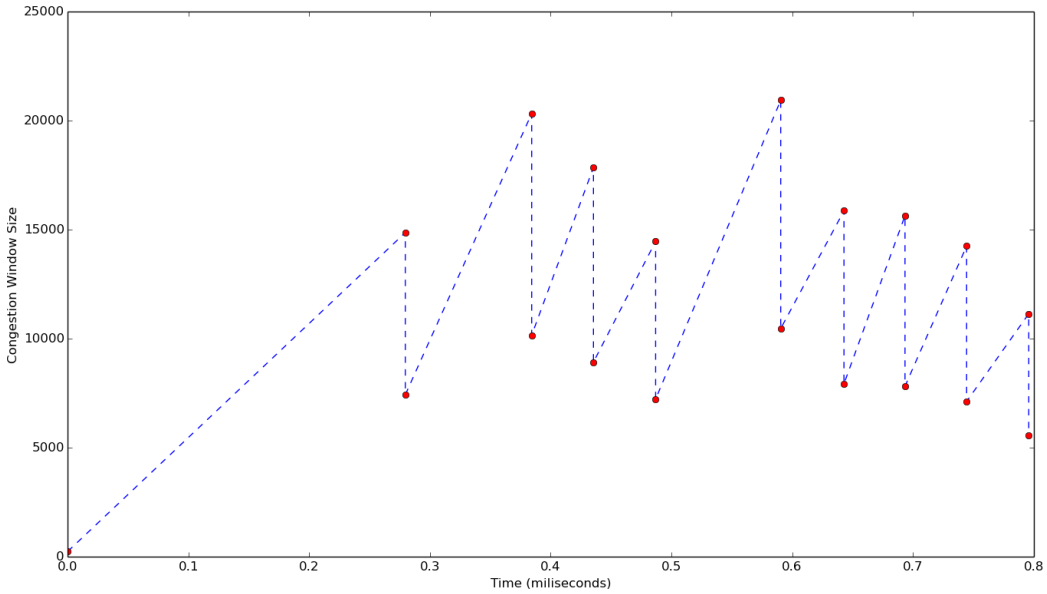


This behavior can be explained as follows:

- When the cumulative ACK timeout is less, throughput is less because more time is spent in thread context switches.
- As we increase timeout, throughput increases because less time is wasted in thread context switches.
- After a threshold value, throughput again starts decreasing because in the case of packet loss, sender now has to wait for a longer time resulting in less bandwidth utilization.

Best throughput that we get with TCP over UDP implementation in user space is ~1.4-1.5Gbps.

We have implemented Go-Back-N protocol at TCP sender which sends all the unacknowledged packets in the case of packet loss which is detected by missing acknowledgement. We have also implemented congestion control mechanism which halves the congestion window size in the case of packet loss (which indicates network congestion) and increases it linearly thereafter. Following is the variation of congestion window size with time while sending around 1Gb data:



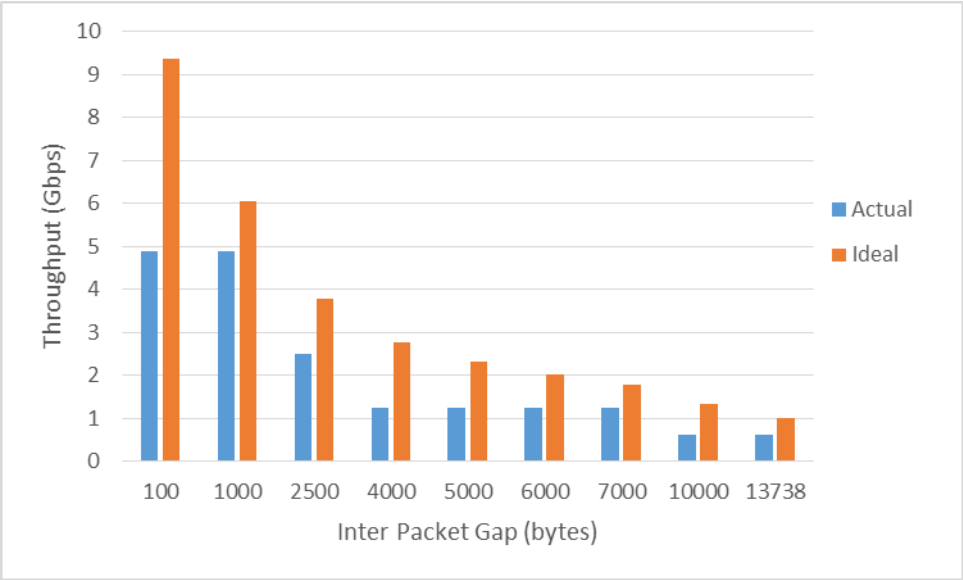
This behavior of congestion window size is as expected and in accordance with RFC.

2.5. TCP over SoNIC: User Space

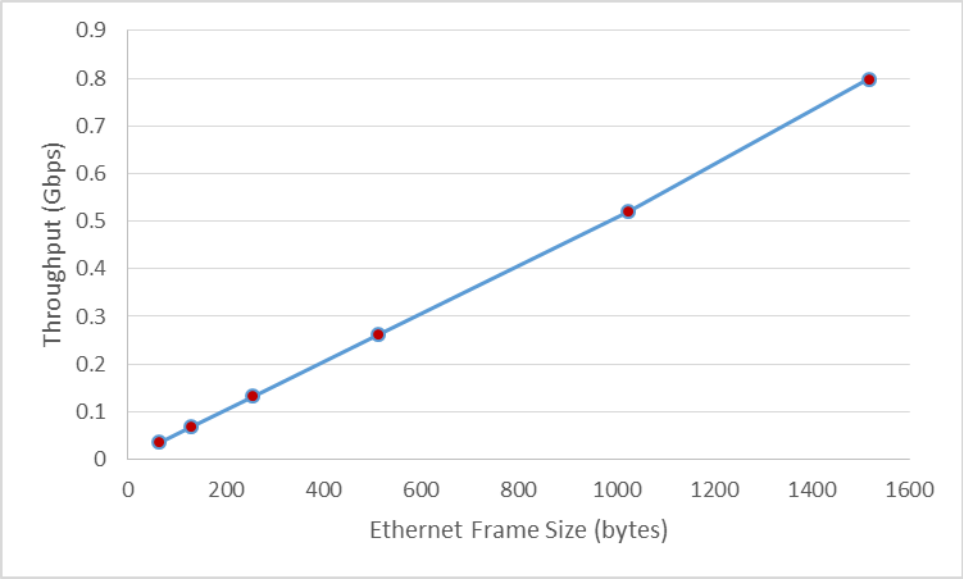
Earlier we decided to skip this step and directly port our TCP over UDP implementation to SoNIC kernel space but we ended up crashing kernel too many times and decided to make it work in user space first. We have created two user space FIFO queues which are shared between Tx and Rx threads of sender and receiver respectively. Below is the code snippet showing this queue sharing:

```
static void sonic_set_fifo_tcp (struct sonic_port * port) {

    struct sonic_fifo *tx_fifo = port->fifo[0];
    struct sonic_fifo *rx_fifo = port->fifo[1];

#if !SONIC_KERNEL
#define SHARED_QUEUE 2
    struct sonic_fifo * pipe = port->fifo[SHARED_QUEUE];
    struct sonic_port * other_port = port->sonic->ports[port->port_id ? 0:1];
#undef SHARED_QUEUE
#endif /*SONIC_KERNEL */

    port->tx_mac->out_fifo = tx_fifo;
    port->tx_pcs->in_fifo = tx_fifo;

    port->rx_pcs->out_fifo = rx_fifo;
    port->rx_mac->in_fifo = rx_fifo;

#if !SONIC_KERNEL
    port->tx_pcs->out_fifo = pipe;
    other_port->rx_pcs->in_fifo = pipe;
#endif
}
```

We measured the throughput by varying the inter packet gap (number of idle bits between successive Ethernet frames) and got the following results:



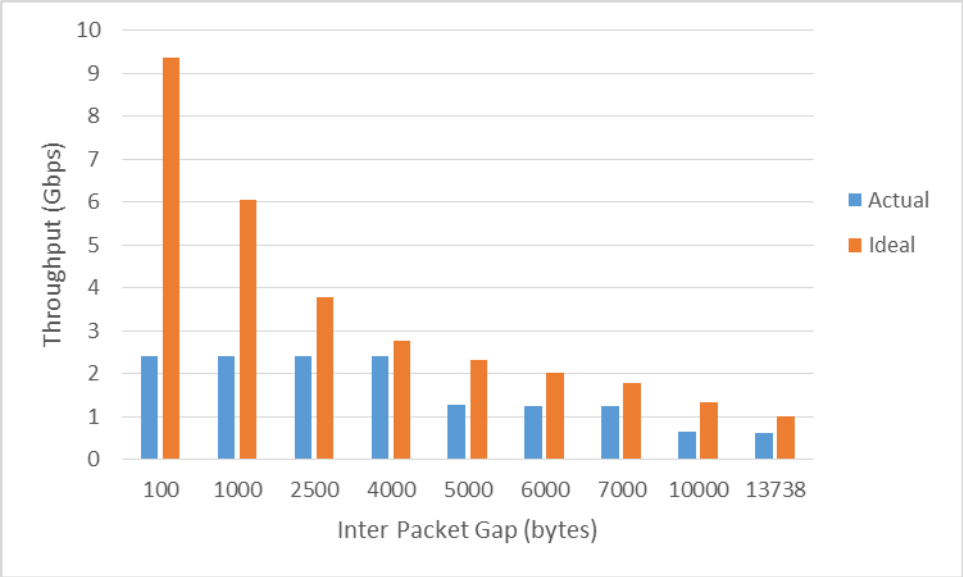
Throughput decreases as we increase the Inter Packet Gap because more and more channel bandwidth is consumed by idle packets. We got maximum throughput of 4.88 Gbps with inter packet gap of 100 bytes. We also measured throughput by varying Ethernet frame size and got the following results:



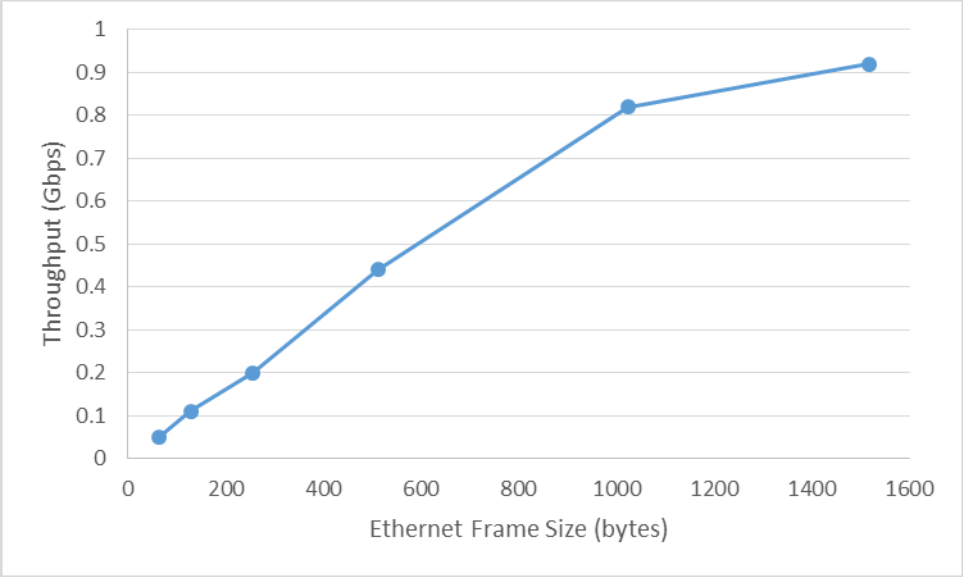
Throughput increases by increasing Ethernet frame size as increasing frame size results in more bandwidth utilization for sending data. We got maximum throughput of 0.79 Gbps with frame size of 1518 bytes.

2.6. TCP over SoNIC: Kernel space

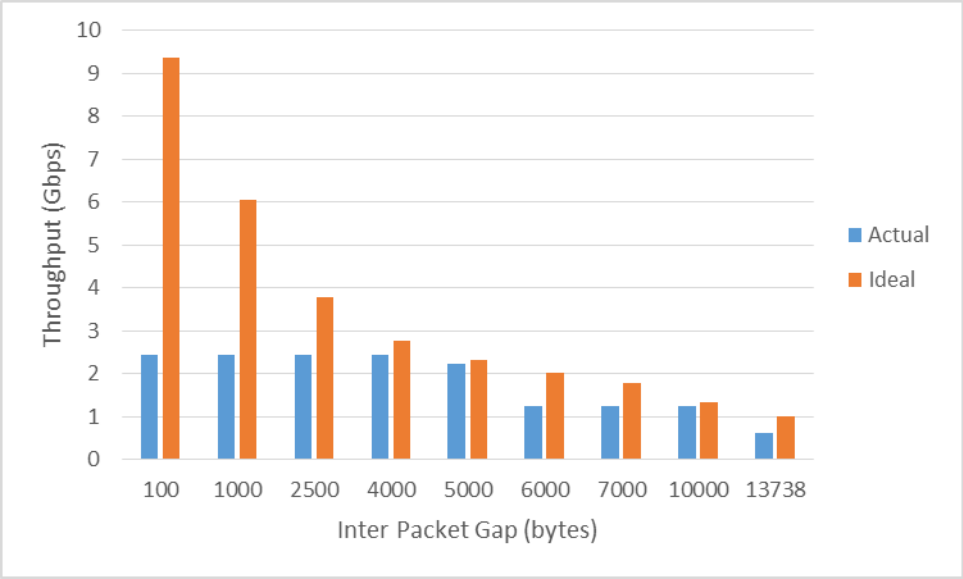
We also ported this TCP implementation to kernel space. We are using atomic variables for synchronization and idle packet count for timeout detection. If we keep on receiving idle packets for some amount of time, we treat this as timeout condition. We ran our implementation in loopback mode in kernel space and got the following results while varying inter packet gap:



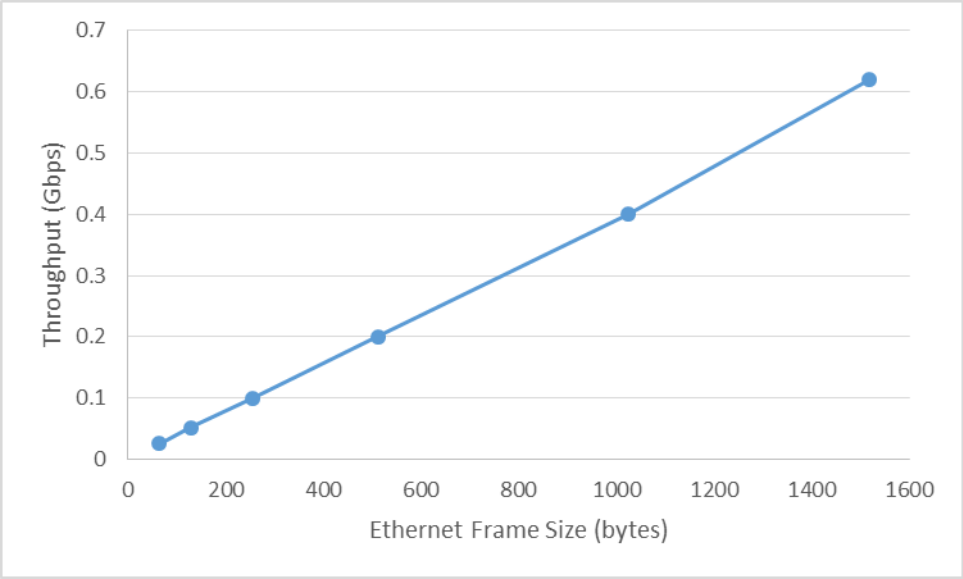
We got maximum throughput of 2.42 Gbps with inter packet gap of 100 bytes. We also measured throughput with different Ethernet frame sizes keeping inter packet gap 13738 bytes and got the following results:



We got maximum throughput of 0.92 Gbps with Ethernet frame size of 1518 bytes. We also made our implementation work across SoNIC cards and got the following results while varying inter packet gap:



We got a maximum throughput of 2.44 Gbps with inter packet gap of 100 bytes. The graph below shows throughput measurements with different Ethernet frame sizes and inter packet gap of 13738 bytes:



We got maximum throughput of 0.62 Gbps with Ethernet frame size of 1518 bytes.

3. Future Work

3.1. Fast Retransmit

TCP implements fast retransmit in order to quickly recover from one packet loss within a stream. We are currently detecting packet loss via timeout while waiting for acknowledgement. We need to implement packet loss detection via 3 duplicate ACKs and then perform fast recovery. The

idea behind assuming packet loss on receiving 3 duplicate ACK is that a sender often sends a large number of segments back to back and if one segment is lost, there will be many back to back duplicate acknowledgements.
We essentially need to implement a Finite State Machine for tracking TCP congestion state – slow start, congestion avoidance and fast recovery.

3.2. Communication with general TCP socket

Current implementation of TCP over SoNIC is capable of communicating with another instance of the same implementation running on the same or different machine. A good extension to it can be to make it capable of communicating with a general TCP socket.

4. References

[1] SoNIC http://fireless.cs.cornell.edu/sonic/

[2] SoNIC: Precise Real-time Software Access and Control of Wired Networks
http://fireless.cs.cornell.edu/sonic/sonic_nsdi2013.pdf

[3] PHY Covert Channels: Can you see the Idles?
http://fireless.cs.cornell.edu/publications/chupja_nsdi2014.pdf

[4] TCP RFC https://www.ietf.org/rfc/rfc793.txt

5. Appendix

5.1. TCP State Machine Implementation

```
#include "tcp_sm.h"
#include "tcp_hdr_util.h"
#include "sonic.h"

/* Transition Functions */
static tcp_state_t no_op( tcp_socket_t* socket, packet_data_t* packet )
{
	//noop
	return CLOSED;
}

static tcp_state_t send_syn( tcp_socket_t* socket, packet_data_t* packet )
{
	socket->snd_nxt += 1;
	return SYN_SENT;
}

static tcp_state_t syn_ack_in_syn_sent( tcp_socket_t* socket, packet_data_t* packet )
{
	if( packet->ack != socket->snd_nxt )
	{
		//incorrect ack?
		return CLOSED;
	}

	socket->rcv_nxt = packet->seq + 1;
	socket->snd_wnd += (packet->ack - socket->snd_una);
	socket->snd_una = packet->ack;
	return SYN_SENT;
}

static tcp_state_t ack_in_syn_sent( tcp_socket_t* socket, packet_data_t* packet )
{
	socket->snd_nxt += 1;
	return ESTABLISHED;
}

static tcp_state_t syn_in_listen( tcp_socket_t* socket, packet_data_t* packet )
{
	socket->rcv_nxt = packet->seq + 1;
	return SYN_RCVD;
}

static tcp_state_t syn_ack_in_syn_rcvd( tcp_socket_t* socket, packet_data_t* packet )
{
	socket->snd_nxt += 1;
	return SYN_RCVD;
}
```

```
static tcp_state_t ack_in_syn_rcvd( tcp_socket_t* socket, packet_data_t* packet )
{
	if( packet->ack != socket->snd_nxt )
	{
		//incorrect ack?
		return CLOSED;
	}

	socket->rcv_nxt = packet->seq + 1;
	socket->snd_wnd += (packet->ack - socket->snd_una);
	socket->snd_una = packet->ack;
	return ESTABLISHED;
}

static tcp_state_t ack_in_established( tcp_socket_t* socket, packet_data_t* packet )
{
	if( packet->dir == SEND )
	{
		socket->snd_nxt += (packet->pkt_count);
	}
	else
	{
		/* accept just next packet. In case of missing pkt discard all
		 * subsequent packets thereby enforcing sender to re-send */
		if( packet->seq == socket->rcv_nxt)
		{
			socket->rcv_nxt = packet->seq + 1;
		}
		if( packet->ack - 1 >= socket->snd_una )
		{
			socket->snd_wnd += (2*(packet->ack - socket->snd_una));
			socket->snd_una = packet->ack;
		}
	}

	return ESTABLISHED;
}

/* Transition Lookup Table */
typedef tcp_state_t (*state_transition_fnptr_t)(tcp_socket_t*, packet_data_t*);
typedef struct {
		tcp_state_t next_state;
		state_transition_fnptr_t action;
	} tcp_transition_t;

tcp_transition_t tcp_state_machine[MAX_STATES][MAX_PKT_TYPES] = {
					{	//closed
						{ CLOSED, send_syn }, //syn
						{ CLOSED, no_op }, //fin
						{ CLOSED, no_op }, //rst
						{ CLOSED, no_op }, //ack
					},

					{	//listen
						{ CLOSED, syn_in_listen },
						{ CLOSED, no_op },
```

```
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
        },
        {       //syn_rcvd
                { CLOSED, no_op },
                { CLOSED, syn_ack_in_syn_rcvd },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, ack_in_syn_rcvd },
        },
        {       //syn_sent
                { CLOSED, no_op },
                { CLOSED, syn_ack_in_syn_sent },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, ack_in_syn_sent },
        },
        {       //established
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, ack_in_established },
        },
        {       //fin_wait_1
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
        },
        {       //fin_wait_2
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
        },
        {       //closing
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
        },
        {       //time_wait
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
                { CLOSED, no_op },
        },
        {       //close_wait
                { CLOSED, no_op },
                { CLOSED, no_op },
```

```
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                        },
                                        {       //last_ack
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                                { CLOSED, no_op },
                                        },
                                };

/* handle packet -- change the state of tcp state machine accordingly */
void process_packet( tcp_socket_t* socket, packet_data_t* packet )
{
        packet_type_t pkt_type = tcp_get_packet_type(packet->flags);
        socket->state = tcp_state_machine[socket->state][pkt_type].action( socket, packet );
}
```