Efficient UDP-Based Congestion Aware Transport for Data Center Traffic

Lisha Ye*, Lotfi Mhamdi§ and Mounir Hamdi*
*Department of Computer Science and Engineering Hong Kong University of Science and Technology, Hong Kong
Email: {lisaye, hamdi}@cse.ust.hk
§School of Electronic and Electrical Engineering, University of Leeds, UK
Email: l.mhamdi@leeds.ac.uk

***Abstract*—Modern Data Centers (DCs) host hundreds of thousands of servers running diverse applications and services. The variety of these applications mandates distinct requirements such as latency and throughput. The state-of-the-art TCP protocol fails to meet these requirements, rendering the design of efficient DC transport protocols an urgent need. Since UDP is the most popular protocol besides TCP, it is a potential alternative to address this challenging problem. In this paper, we propose DCUDP, a UDP-like protocol for DCs which provides excellent congestion control using Explicit Congestion Notification (ECN). DCUDP achieves excellent throughput both for normal and short flows through simulations. Furthermore, DCUDP guarantees fairness and convergence during periodic congestions (burstiness). Our experiments show that DCUDP outperforms many TCP-like protocols for data centers in terms of throughput, fairness and convergence.**

***Index Terms*—Data center, UDP, Active Queue Management, Throughput, Fairness.**

I. Introduction

The exponential growth in cloud computing is stressing the need for the design of scalable and efficient Data Centers (DCs) capable of sustaining the current and projected increase in cloud services and applications. Modern DCs host diverse applications with various stringent performance requirements in terms of latency, throughput, Quality of Services (QoS), etc. In order to meet the distinct requirements of these applications, careful design considerations have to be given to the Date Center Network (DCN) infrastructure and its transport layer protocol. Unfortunately, the standard TCP falls short in satisfying current DCN traffic in many aspects including bandwidth utilization, congestion as well as fairness, all of which greatly affect the performance and scalability of the DCN. Meanwhile, for data centers, cost is also a critical problem faced with large-scale hardware upgrade and deployment. To strike balance in both performance and cost, how to optimize current traffic protocols for future DCNs at a reasonable price has become a challenging issue.

Due to the nature of TCP, current TCP including TCP-like protocols in data centers cannot suit the data center traffic characteristics especially in aspects like utilization, latency and fairness. Faced with RTT as small as $100\mu s$ to $300\mu s$ in data centers [2], the "slow-start" phase and the RTT-driven nature of TCP may delay flows especially short and emergent ones, resulting in missing deadlines. During a "slow-start" phase, it was shown [3] that many flows finish transmission even before the "slow-start" phase ends although the bandwidth is under-utilized. Also, as most edge switches are shallow-buffered, TCP is too dumb for congestions with limited buffer space, leading to TCP time-out and therefore to the "incast" problem [4][5], mainly due its coarse-grained Minimum Retransmission Timeout (RTO_{min}) [6]. As a result and because of the shortcomings of TCP, various other protocols are being studied and proposed for better transmission in DCs. A potential candidate of these protocols is UDP as a viable alternative to current TCP.

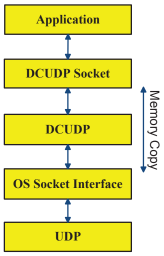


Fig. 1. DCUDP Architecture. DCUDP builds a middle layer between application and OS socket interfaces and the core DCUDP transmits through UDP

In this paper, we describe DCUDP, an ECN-capable UDP-like protocol for data center that guarantees optimal throughput and congestion avoidance. Inspired by previous research in UDP [7][8][9], DCUDP supports reliable transmission and do not require modifications on traditional UDP. When the bandwidth is underutilized, it sends packets as much as possible (UDP Mode) which greatly improves work-flow sending rate compared to TCP. On the other hand, when the bandwidth is not enough and switch queue becomes growing (Congestion Mode), DCUDP react much faster than other UDPs faced with sudden burstiness. Additionally, DCUDP is designed to provide better fairness than other TCP-like protocols for data centers during concurrent Map-Reduce flow transmissions, each of which has various RTT.

The rest of this article is structured as follows: Section II briefly reviews the related work on existing protocols designed for data centers. We then describe the design and algorithm of DCUDP in Section III and IV. In Section V, we test DCUDP and existing TCP-like protocols for data centers using the NS-

978-1-4799-2730-2/14/$31.00 ©2014 IEEE

2 simulator [10] under Map-Reduce like traffic settings with various RTTs. We show the superior performance of DCUDP compared with other protocols in terms of throughput, fairness and convergence. Finally, we conclude our work in Section VI.

II. Related Work

The transport layer design for DCNs evolved from the traditional TCP protocol. Unfortunately, standard TCP fails in DCN contexts due to its fundamental drawbacks in such environment. Because of the shortcomings of TCP, numerous DCN protocols have been proposed. Various proposals employ implicit rate control mechanisms, such as congestion control and notification algorithms. Examples of these protocols include DCTCP [2], D^2TCP [11] and HULL [12]. These approaches rely on carefully estimated flow rates while maintaining high network utilisation. This is fundamentally challenging due to the highly dynamic DCN workloads and network continuous utilization. Other protocols such as RCP [13], QCN [14] and D^3 [3] aimed at providing much adequate and fair bandwidth allocation for workloads, thus providing better performance both in presence of congestion and idleness. However, the expense on device upgrade on edge switches and end servers is too high for current networks. Other researchers have proposed UDP-like protocols for high-bandwidth low-latency networks, such as UDT [8], DCCP [15] and RBUDP [7]. However, most of these proposals aim at providing unreliable traffic, which is unacceptable for most DCN applications. More recent research work focused on proposing more reliable DCN transport design, such as DeTail [16], PDQ [17] and pFabric [18]. Despite the high potential of these proposals, they have ignored the DCN node (switches and routers) architectural impact on their design by just assuming traditional switch design proposed for earlier communications Networks, which may prove challenging in a DCN environment.

DCTCP is the most closely related work to our proposal. DCTCP is the best AQM-enabled TCP-like protocol proposed that can control queue size to a small threshold while maintaining throughput. Unfortunately, DCTCP still fails in some aspects. First, as has been stated before, DCTCP inherits the nature of TCP which sometimes greatly affect bandwidth utilization. Secondly, the unfairness problem is even worse for DCTCP than traditional TCP or ECN-enabled TCP. DCTCP drops and marks packets with full probability based on the instantaneous queue size rather than the average queue size, and it calculates the cut of congestion window level based on the portion of ECN-echo ACK received during the last window of packet number transmitted (roughly the number of ECN-echo ACK packets/last congestion window size). This mechanism is sensitive to sense congestion but also sometimes too radical in congestion control, resulting in severe unfairness towards flows that have just started when the queue size has exceeded the threshold. Combined with RTT-unfairness problems induced by TCP nature, DCTCP suffers from sort of unfairness problems which may delay application completion time at the price of better queue management. As we shall demonstrate later in this article, although our proposed DCUDP also uses ECN marking, it guarantees better fairness than DCTCP.

III. DCUDP Design

A. Background of UDT

The proposed DCUDP is inspired by UDP-based Data Transfer (UDT) [8] in reliable transmission. Here we introduce UDT first.

1) Architecture: UDT adapts into the layered network protocol architecture and uses UDP through socket communication provided by the operating systems. Applications pass data through a UDT socket which uses UDP for sending and receiving data. DCUDP inherits this architecture (Figure 1).

2) Reliable Transmission: UDT is a connection-oriented duplex so each UDT entity has a pair of sender and receiver, with data flows sent from the sender to the receiver and control flow exchanged between two receivers. There are two types of packets in UDT: data packets and control packets. UDT uses control packets to support reliable transmission. Here we introduce the three critical control packets that serve for reliability.

- **ACK & ACK2**: UDT does not use ACK at the sender side, but uses a pair of ACK and ACK2. The receiver of UDT periodically sends ACK and the sender side sends back ACK2 for RTT calculation. Based on the RTT, UDT executes congestion control.
- **NAK**: UDT uses NAK for packet loss signalling. As long as the receiver gets inconsequent sequence numbers in data packets, it sends NAK at once to the sender. NAK contains “control information” to notify the senders which packets are lost for retransmission.

Other control packet types include: Hand-shake, keep-alive, shutdown and so on. DCUDP inherits all control packet types.

B. Packet Structure of DCUDP

For DCUDP (Figure2), similar to UDT, the first bit helps distinguish data and control packets. Data packets contain a sequence number, a message sequence number and a relative timestamp. For the control packets, the type of information is put in the bit field. Each control packet is assigned a unique 31-bit sequence number, independent of data packets. Both data packets and control packets have an Observe (OBS) bit for observing which of the transmission modes is being used. To support ECN, the Congestion Window Reduced (CWR) and ECN-Echo (ECE) bits are added.

- **OBS bit**: DCUDP has two modes for transmission. The OBS is set to 0 during UDP Mode and if the receiver receives CE code point, the OBS bit is set to 1 to notify the senders entering Congestion Mode for congestion control. To synchronize the OBS bit change, an OBS-change control packet is used.
- **CWR and ECE bits**: The ECE and CWR bits work in the same way as TCP-ECN [19]. The process of this is that: If the receiver gets a data packet with CE code point set on the IP layer, the receiver sends ACK with ECE bit

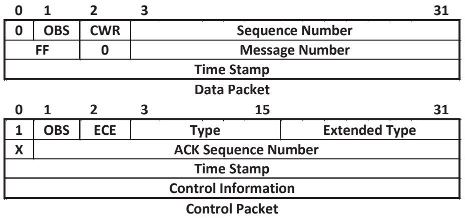


Fig. 2. The DCUDP Packet Header.

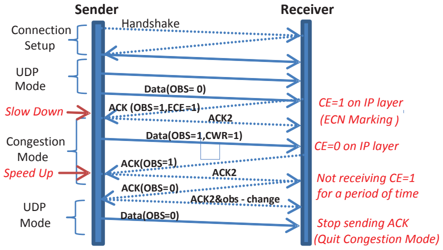


Fig. 3. The DCUDP Transmission Procedure.

set. When the sender finds ECE bit set, the sender slows down and sets CWR bit to notify that the last ECN echo is already received.

C. Tansmission Procedure of DCUDP

1) Connection Setup: The setup of connection is a client/server mode handshake process which is the same as that of UDT.

2) Transmission Mode Switch: As can be seen from Figure 3, the default mode is UDP Mode. In this mode, the sender sends out data packets within a UDP way. The receiver does not send ACK, but keep-alive packets from time to time. The sender enters Congestion Mode when the OBS bit is set and then ACK packets are sent every RTT. Once the sender gets ACK, it replies with an ACK2 and reacts as follows:

- If the ECE bit is not set, it speeds up the sending rate.
- If the ECE bit is set, it slows down the sending rate. If the sender gets NAK, it also slows down.

If the receiver does not sense congestions for a relatively *long* time, the receiver sends ACK packet with $OBS = 0$ to notify the sender to quit the Congestion Mode.

3) Tear Down: When one of the peers is closed, it sends a shutdown message for notification. This also works similar to UDT.

IV. DCUDP Algorithm

In this section, we introduce the algorithms of DCUDP during congestions. We assume the bottleneck link capacity as C (Gbps) and the average packet size as $PktSize$ (KB).

A. Congestion Mode

DCUDP enters the Congestion Mode using an OBS bit change, which is triggered by the following logic:

1) Switch Side: The CE code marking at the IP layer is based on two threshold th_min and th_max, same as TCP-ECN. For DCUDP we set th_min as 1 (Queue in packets). As the queue size grows to th_max, the switch marks with uniform probability. For the queue size observation, we use instantaneous size rather than average size, so that DCUDP can be sensitive to burstiness. When the queue size grows larger than th_max, we do not drop packets, in order to prevent huge packet loss during UDP Mode.

The choice of th_max is important because it needs time to control the queue caused by a greedy UDP transmission before entering the Congestion Mode. So th_max cannot be too large or it will be not be suitable for congestions. Assume:

- The maximum switch buffer size as Max_qsize,
- The average sending interval during UDP Mode as $Interval$,
- The average link delay as $Delay$,
- The RTT during the idle times as RTT_{idle},
- The expected maximum number of senders as N.

Choose th_max as long as the following equation stands:

$$\begin{aligned} th\_max \quad &+ \quad N \times (\frac{RTT_{idle} + th\_max \times Delay}{Interval}) \\ &- \quad \frac{RTT_{idle} + th\_max \times Delay}{Delay} \\ &\leq \quad Max_{qsize} \end{aligned}$$

Note that, with N servers sending concurrently in a bottleneck link and even if the queue length exceeds th_max and all servers are under the UDP Mode, the equation above ensures tolerable packet loss.

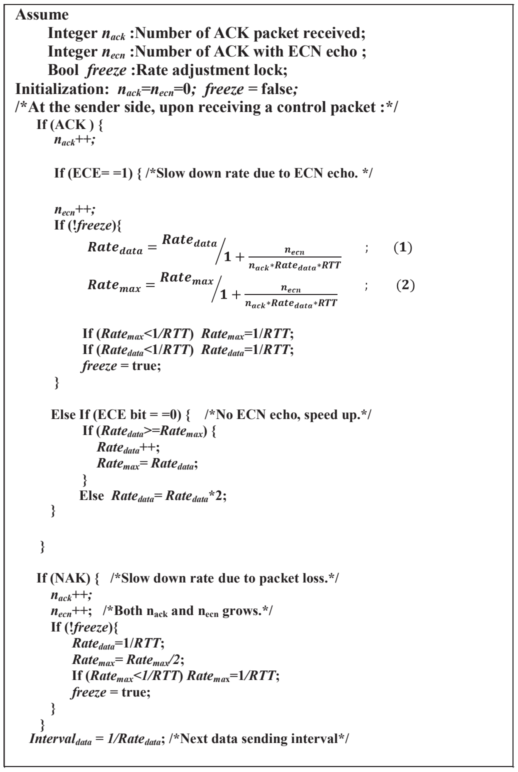
2) Receiver Side: During UDP Mode, the receiver side does not send ACK but watches the CE bit of data packets. If the receiver gets the CE bit set, it immediately sends an ACK to the sender with the OBS bit set. As the receiver gets the $ACK2$ reply, it calculates recent RTT based on the peers. The $ACK2$ reply is set with the same sequence number as its ACK peer:

$$\begin{cases} RTT_{current} = Timestamp_{ACK2} - Timestamp_{ACK} \\ RTT = (1-\alpha) \times RTT + \alpha \times RTT_{current}; \ \ (\alpha = 0.125) \\ Interval_{ACK} = RTT \end{cases}$$

After that the receiver starts sending ACK packets every RTT and the initial value is RTT_{idle}. The receiver side updates RTT each time it receives $ACK2$.

3) Sender Side: When the sender receives the first ACK, it enters the Congestion Mode and replies by $ACK2$. For the mode switch:

- **Step 1**: Set OBS bit to 1 in the following data packets.
- **Step 2**: Slow down the current sending interval to $Interval_{data}$. Here we have two rates for $Interval_{data}$ calculation: $Rate_{data}$ and $Rate_{max}$. The latter one is the maximum $Rate_{data}$ a sender is allowed to achieve during



```
Assume
        Integer $n_{ack}$ :Number of ACK packet received;
        Integer $n_{ecn}$ :Number of ACK with ECN echo ;
        Bool $freeze$ :Rate adjustment lock;
Initialization: $n_{ack}$=$n_{ecn}$=0; $freeze$ = false;
/*At the sender side, upon receiving a control packet :*/
    If (ACK ) {
        $n_{ack}$++;

        If (ECE= =1) { /*Slow down rate due to ECN echo. */

        $n_{ecn}$++;
        If (!$freeze$){
            $Rate_{data} = Rate_{data} / 1 + \frac{n_{ecn}}{n_{ack}*Rate_{data}*RTT}$    ;    (1)
            $Rate_{max} = Rate_{max} / 1 + \frac{n_{ecn}}{n_{ack}*Rate_{data}*RTT}$    ;    (2)

            If ($Rate_{max}$<1/$RTT$)  $Rate_{max}$=1/$RTT$;
            If ($Rate_{data}$<1/$RTT$)  $Rate_{data}$=1/$RTT$;
            $freeze$ = true;
        }

        Else If (ECE bit = =0) {    /*No ECN echo, speed up.*/
            If ($Rate_{data}$>=$Rate_{max}$) {
                $Rate_{data}$++;
                $Rate_{max}$= $Rate_{data}$;
            }
            Else  $Rate_{data}$= $Rate_{data}$*2;
        }

    }

    If (NAK) {   /*Slow down rate due to packet loss.*/
        $n_{ack}$++;
        $n_{ecn}$++;   /*Both $n_{ack}$ and $n_{ecn}$ grows.*/
        If (!$freeze$){
            $Rate_{data}$=1/$RTT$;
            $Rate_{max}$= $Rate_{max}$/2;
            If ($Rate_{max}$<1/$RTT$) $Rate_{max}$=1/$RTT$;
            $freeze$ = true;
        }
    }
  $Interval_{data}$ = 1/$Rate_{data}$; /*Next data sending interval*/
```

Fig. 4. The Congestion Control Algorithm of DCUDP.

the Congestion Mode.

$$\begin{cases} Rate_{data} = \frac{C \times 10^6}{8 \times PktSize}(Packet/sec) \\ Interval_{data} = \frac{1}{Rate_{data}}(sec) \\ Rate_{max} = Rate_{data} \end{cases}$$

Notice that both $Rate_{data}$ and $Rate_{max}$ are each at least one data packet per RTT for fairness concern.

B. Congestion Control during Congestion Mode

In the Congestion Mode, the congestion control algorithm is applied on the sender through the adjustment of $Interval_{data}$.

Here we introduce the algorithm (Figure 4) at the sender. There are three types of control packets received at the sender during the Congestion Mode with the corresponding reactions. For the Congestion Control Algorithm, note that:

- On receiving ECN-echo (ECE), unlike TCP, we do not halve $Rate_{max}$ and $Rate_{data}$ but cut in a smoother way (see Equations (1) and (2) in Figure 4). It is because the sending interval of ACK is now RTT. In a high-speed environment, definitely more than one data packet is received during an RTT, so the likelihood of rate increase in DCUDP is less than in TCP.
- We use $\frac{n_{ecn}}{n_{ack}}$ for congestion level signal, which is the portion of ECN-echo ACK received during a Congestion Mode period.
- We use $freeze$ as a rate adjustment lock. We do not want senders that receive huge ECN-echo packets to slow down too much at once. Each time the sender sends out a data packet after $Interval_{data}$, it sets $freeze$ as false.

C. Quit Congestion Mode

The receiver side observes the end of the congestion period and senders reenter the UDP Mode after the congestions.

1) Receiver Side: The receiver uses functions $getDelayTrend()$ and $getEcnTrend()$ every time an $ACK2$ is received. We use several windows of size K for recent control message storage (recent RTT and the percentage of data packets with CE set). The oldest message is replaced after the windows are full. K is chosen by the average congestion length and RTT in real practice.

- **getDelayTrend():** Each latest RTT calculated is stored in $rttWindow$. Among all recent $RTTs$ recorded, the function calculates and returns the percentage (P) of the RTT that is smaller than the previous one ($rttWindow(i\%K) \leq rttWindow((i-1)\%K)$):

$$P_{RTT \leq RTT'} = \frac{Number\ of\ RTT \leq RTT'}{Number\ of\ RTT} \quad (3)$$

- **getEcnTrend():** For every data packet received, the receiver checks the CE bit and determines whether to return ECN-echo. Therefore, for every K recent data packets received, the receiver records the portion (p_{ecn}) of the packets with the CE bit set in a p_{ecn} Window of size K. When K portions are stored, calculate the percentage of the p_{ecn} that is smaller than the previous one ($p_{ecn}Window(i\%K) \leq p_{ecn}Window((i-1)\%K)$). The window stores the ECN trends of at most K^2 recent packets received.

$$P_{p_{enc} \leq p'_{enc}} = \frac{Number\ of\ p_{enc} \leq p'_{enc}}{Number\ of\ p_{enc}} \quad (4)$$

Equations (3) and (4) cope to sense congestion ending. The logic is described as follows:

- **Condition 1:** If the following condition is satisfied, the ACK sending interval is enlarged to a fixed interval as $\beta \times RTT_{idle}$.

$$If \begin{cases} \mathcal{P}_{p_{enc} \leq p'_{enc}} < \theta_{ecn}\ AND \\ \mathcal{P}_{RTT \leq RTT'} < \theta_{RTT}\ AND \\ latestP_{ecn} = 0 \end{cases}$$

Then $Interval_{ACK} = \beta \times RTT_{idle} \quad (\beta \geq 1)$[1]

- **Condition 2:** If Condition 1 is satisfied, then quit the congestion Mode if the following condition is satisfied.

$$If \begin{cases} P_{p_{enc} \leq p'_{enc}} = 1\ AND \\ P_{RTT \leq RTT'} = 1\ AND \\ latestP_{ecn} = 0 \end{cases}$$

Then set OBS bit = 0 for the following ACK sent.

If Condition 2 is satisfied, the receiver sends ACK with $OBS = 0$. Once the receiver gets an OBS-change signal from the sender, it stops sending ACK. If condition 2 is not satisfied but new ECN arrives again, $Interval_{ACK}$ is reset to RTT.

[1] In our simulation, we set as θ_{ecn} to 0.5, β to 2 and θ_{RTT} to 0.2.

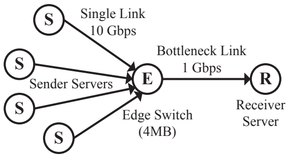


Fig. 5. The Topology of Experiments

2) Sender Side: If the sender gets control packets with $OBS = 0$, it first sends an OBS-change signal, then changes OBS bit to 0 for the following data packets and quits the Congestion Mode.

V. Simulation Results

In this section, we use the NS-2 simulator to compare DCUDP with other protocols for data centers in terms of throughput during and after congestion periods.

A. Throughput under highly congested scenarios

In this experiment, we measure the throughput during both regular (1s) and short (10ms) flows during congested concurrent traffic flows. We use several sender machines connected to an edge switch (4 MB buffer size [2]) with 10 Gbps capacity as depicted in Figure 5). A receiver machine is connected to the switch with 1 Gbps capacity. Each sender sends constant bit rate traffic (CBR) at 1.6 Gbps to the receiver lasting for 1s and 10ms respectively. The RTT_{idle} for each link is 100 μs. We use DCUDP ($th_max = 15$), DCTCP ($g = 0.3$, $K = 10$ which is a recommended choice) and TCP Reno for comparison. For TCP, the queue is drop-tail while the others are RED ($gentle = false$) with $ecn_{bit} = true$. UDT causes intolerable packet loss in the experiment so we do not discuss it.

We find that, for the case of normal-sized flows (1s) as in Figure 6(a) and with servers from 1 to 20, both DCTCP and DCUDP achieve a throughput as high as 0.98 Gbps, while TCP falls behind a lot. It again proves that traditional TCP produces enormous RTO (compared to RTT_{idle}) under highly congested situations [20][4].

For a short flow case in Figure 6(b), DCUDP achieves better throughput than the other two since it does not have a "slow-start" phase. Interestingly, DCTCP performs even worse than traditional TCP here due to its overreacting congestion control. For short and emergent flows, this is intolerable.

To observe the congestion control performance, we compared the queue length of TCP, DCTCP and DCUDP under normal sized flows of 1s using 20 servers as illustrated in Figure 7(a). Note that similar experiments on TCP-ECN have been done in [2], we therefore do not plot its throughput and queue size for simplicity. We find that traditional TCP is poor in both throughput and queue length. DCUDP gets heavy queue burden at the beginning of congestion, and then quickly drops to a stable small value, as can be seen from Figure 7(b). DCTCP is the best in queue length control among all including TCP-ECN, although the cost of effective queue control is overreaching at times, especially for short flows. This proves DCTCP effective in queue control during burstiness with similar throughput.

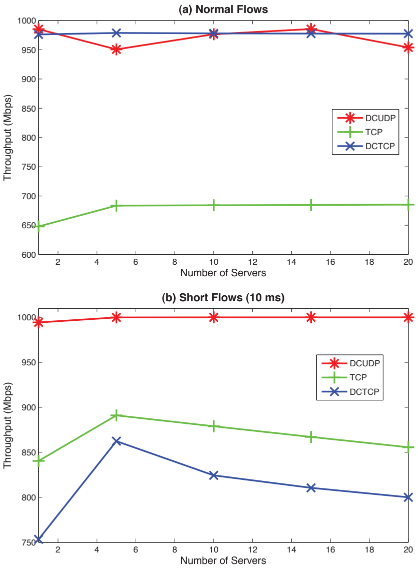


Fig. 6. Throughput under congestions: (a) Norml Flows (1s). (b) Short Flows (10 ms)

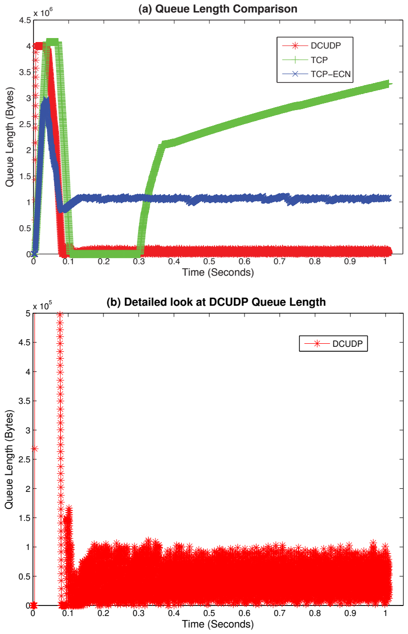


Fig. 7. Queue Length Variation: (a) Queue length variation. (b) DCUDP Queue length variation.

B. Immediate Throughput after Congestion

Here we use the previous topology but this time we use only two senders. One sender sends CBR traffic at 1 Gbps, and the other sends File Transfer Protocol (FTP) traffic. Both senders start at Time (0s), but the FTP traffic finishes at 0.5s and the CBR traffic stops at 1s. We test whether the CBR traffic

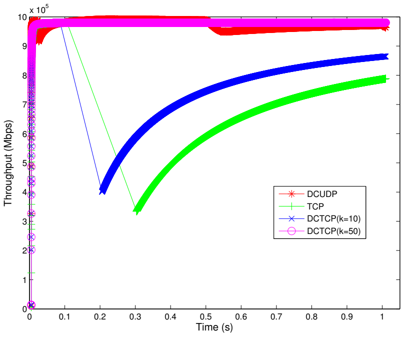


Fig. 8. Throughput changes when one flow ceases at Time (s) = 0.5.

can quickly take advantage of the free available bandwidth after the FTP traffic ceases. We use TCP, DCTCP ($K = 10$), DCTCP ($K = 50$) and DCUDP ($th\_max = 15$) to test the throughput variation during 1s and we plot the results in Figure 8.

We find that both DCTCP ($K = 50$) and DCUDP quickly use the free bandwidth after one flow ceases, while TCP and DCTCP ($K = 10$) do not profit from the available bandwidth. The reason is that TCP packet retransmission mechanisms during the previous congestion period make the congestion window grow linearly even if it senses idleness afterwards. Also, for DCTCP with small K (e.g. 10) is similar to TCP. DCUDP does not have this problem because it is not TCP based and it has its own algorithm for sensing the switching from congestion to idleness (Section IV). Figure 9 illustrates the difference in DCTCP congestion window growth with $K = 10$ and $K = 50$ after 0.5s. The congestion window of $K = 50$ gains a sharp growth but $K = 10$ does not.

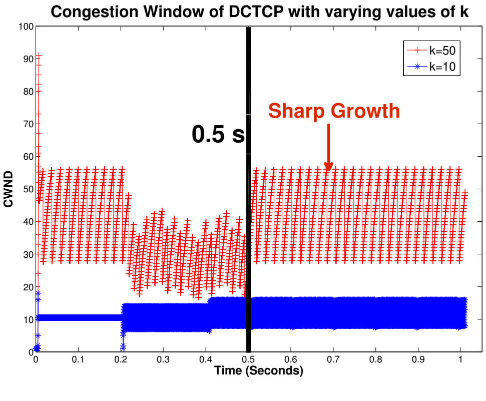


Fig. 9. Congestion window Comparison for DCTCP with varying K values.

VI. Conclusion

In this paper, we have introduced DCUDP, a UDP-like protocol for reliable transmission in DCNs. We first listed current findings in traffic characteristics in DCNs and pointed out the unsuitability of standard TCP in modern DCN environments. We then described the design and structure of DCUDP as well as its algorithms during congestion and idle periods. Finally, we used NS-2 to simulate DCUDP and conduct several experiments which are similar to bursty data center Map-Reduce traffic. With ECN-capability, DCUDP has been proved a viable solution and can not only provide high throughput in data centers, but also strong congestion control during peak periods. Although UDP is not the most prevalent protocol in data center reliable transmission, we show that there is a strong need (and reason) to adopt UDP-like protocols for transport since short flows frequently appear in most current Map-Reduce applications, and sometimes tight deadlines are required but the slow-start phase and congestion control method in traditional TCP cannot fully utilize bandwidth efficiently.

References

[1] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *ACM SIGCOMM’10*, (New York, NY, USA), pp. 63–74, ACM, 2010.

[3] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, “Better never than late: meeting deadlines in datacenter networks,” in *ACM SIGCOMM’11*, (New York, NY, USA), pp. 50–61, ACM, 2011.

[4] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, “Safe and effective fine-grained tcp retransmissions for datacenter communication,” in *ACM SIGCOMM’09*, (New York, NY, USA), pp. 303–314, ACM, 2009.

[5] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, “Understanding tcp incast throughput collapse in datacenter networks,” in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN ’09, (New York, NY, USA), pp. 73–82, ACM, 2009.

[6] V. Paxson and A. M., “Computing tcps retransmission timer.,” *IETF RFC 2988*, November 2000.

[7] E. He, J. Leigh, O. Yu, and T. A. DeFanti, “Reliable blast udp: Predictable high performance bulk data transfer,” in *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER ’02, (Washington, DC, USA), pp. 317–, IEEE Computer Society, 2002.

[8] Y. Gu and R. L. Grossman, “Udt: Udp-based data transfer for high-speed wide area networks,” *Comput. Netw.*, vol. 51, pp. 1777–1799, May 2007.

[9] P. Saab, “Qcn: Quantized congestion notification,” Dec. 2008.

[10] “The network simulator - ns-2.”

[11] B. Vamanan, J. Hasan, and T. Vijaykumar, “Deadline-aware datacenter tcp (d^2tcp),” in *ACM SIGCOMM’12*, pp. 115–126, 2012.

[12] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, “Less is more: trading a little bandwidth for ultra-low latency in the data center,” in *The 9th USENIX conference on Networked Systems Design and Implementation*, NSDI’12, pp. 19–19, 2012.

[13] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, “Processor sharing flows in the internet,” in *IWQoS’05*, (Berlin, Heidelberg), pp. 271–285, Springer-Verlag, 2005.

[14] R. Pan, B. Prabhakar, and A. Laxmikantha, “Qcn: Quantized congestion notification,” 2007.

[15] E. Kohler, M. Handley, and S. Floyd, “Designing dccp: congestion control without reliability,” (NY, USA), pp. 27–38, ACM, 2006.

[16] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, “Detail: reducing the flow completion time tail in datacenter networks,” in *ACM SIGCOMM’12*, pp. 139–150, 2012.

[17] C.-Y. Hong, M. Caesar, and P. B. Godfrey, “Finishing flows quickly with preemptive scheduling,” in *ACM SIGCOMM’12*, pp. 127–138, 2012.

[18] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, “pfabric: minimal near-optimal datacenter transport,” in *ACM SIGCOMM’13*, pp. 435–446, 2013.

[19] K. Ramakrishnan, S. Floyd, and D. Black, “The addition of explicit congestion notification (ecn) to ip,” *IETF RFC 3168*, september 2001.

[20] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, “Measurement and analysis of tcp throughput collapse in cluster-based storage systems,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST’08, (Berkeley, CA, USA), pp. 12:1–12:14, 2008.