# LTTP: An LT-Code Based Transport Protocol for Many-to-One Communication in Data Centers

Changlin Jiang, Dan Li, *Member, IEEE,* and Mingwei Xu, *Member, IEEE*

***Abstract*—TCP has been widely adopted in current data centers to ensure reliable data delivery. However, recently TCP Incast was found to occur in many-to-one communications with barrier-synchronized requirement, where the TCP goodput drops dramatically. Previous solutions to TCP Incast either require updating the OS/hardware to support fine-grained timers, or smartly control utilization of the switch buffer to reduce the probability of buffer overflow and packet loss.**

**In this paper we explore a different approach to support many-to-one communication in data center networks, which we call *LTTP* (LT-code based Transport Protocol). LTTP improves LT (Luby Transform) code to achieve reliable UDP-based transmission by exploiting data redundancy, and employs TFRC (TCP Friendly Rate Control) to adjust the traffic sending rates at servers. NS-2 based simulation shows that the goodput of LTTP never degrades with the increase of the number of servers in many-to-one communications, and LTTP significantly outperforms DCTCP [1] when the number of servers is large. Simulation results also demonstrate that LTTP flows can fairly share bandwidth with TCP flows.**

***Index Terms*—TCP Incast, digital fountain, TCP-friendly, LT code.**

## I. Introduction

CLOUD computing realizes the dream of "computing as a utility". People outsource their computing and software capabilities to cloud providers and pay for the service usage on demand. In the cloud data centers, tens of thousands of, or even hundreds of thousands of servers are interconnected by the network infrastructure, to carry out large-scale distributed computation with SLA (Service Level Agreement) guarantees. In consideration of economical cost, cloud providers tend to use low-end commodity servers and switches to build the data center, and the reliability and performance requirements are met by upper-layer software.

Cloud data centers run both online services, such as search and social network, and back-end computations, such as MapReduce [2] and GFS [3]. Since most distributed computations in data centers are bandwidth-hungry, advanced network topologies, e.g., Fat-Tree [4] and VL2 [5], are proposed to increase the network capacity. In these topologies, a large number of 1GE or 10GE COTS (commodity, off-the-shelf) switches are interconnected to form a high-radix, low-diameter data center network. In a non-blocking network infrastructure, the end-to-end network throughput between servers can be up to 1Gbps and the end-to-end delay is in the order of hundreds of microseconds, which exposes a shifted network paradigm from the wide-area Internet.

As for data transmission between servers, TCP is widely used in today's data center networks, since it has been proven a great success in the Internet for both reliable delivery and congestion control. However, the specific application pattern and network environment in data centers pose new challenges to TCP to work smoothly. A recently found problem is TCP Incast [6], in which TCP experiences severe goodput collapse in the setting of barrier-synchronized many-to-one workloads [7]. In this kind of communication pattern, a client sends requests to multiple servers simultaneously, and the servers then send data blocks back to the client immediately upon the requests. The client will not send the next request until all the requested data blocks have been received.

TCP Incast causes goodput collapse for two reasons. Firstly, when servers simultaneously send response packets back to the client, the response packets will overflow the output buffer of the switch which directly connects the client. Secondly, the default value of TCP's Retransmission Timeout (RTO) is 200 milliseconds in most operating systems. It means that once a timeout occurs, the TCP connections will be idle for quite a long time period before the servers retransmit the dropped packets, since the RTT (Round-Trip Time) is only hundreds of microseconds in data center networks. After the retransmission timer timeouts, the servers will again simultaneously send the response packets, which causes switch buffer overflow and retransmission for a new round, so on and so forth.

The goodput degradation in many-to-one communications will significantly delay the task finish time of distributed computations, which is further translated to the violation of SLA. Since the root cause for TCP Incast is the shallow buffer in switches as well as the mismatch between RTO and RTT, existing solutions focus on either preventing flows from aggressively utilizing the switch buffer, e.g., DCTCP [1] , ICTCP [8], FQCN [9] and AF-QCN [10], or minimizing the RTO value [7], [11]. However, fine-grained RTO (e.g., microseconds) requires updating the OS or hardware, and keeping switch buffer utilization low cannot fundamentally solve the problem when the number of servers is large enough and packet loss indeed occurs.

Different from previous solutions, in this paper we propose a new transport protocol to support many-to-one communications in data centers, which is called *LTTP* (LT-code based Transport Protocol). Since TCP's timeout is the root cause of low link utilization and goodput deterioration in TCP Incast,

Manuscript received January 15, 2013; revised July 1, 2013.

C. Jiang, D. Li and M. Xu are with the Department of Computer Science and Technology, and are with Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China (e-mail: jiangchanglin@csnet1.cs.tsinghua.edu.cn, tolidan@tsinghua.edu.cn, xmw@cernet.edu.cn).

Digital Object Identifier 10.1109/JSAC.2014.140106.

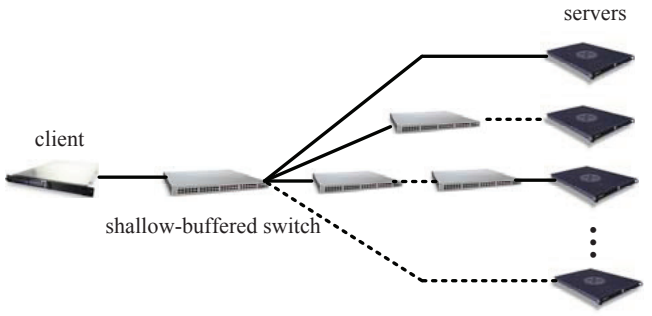



Fig. 1. A typical setup of TCP Incast. Many servers send data to the client simultaneously, which overflows the shallow buffer in the TOR switch.

LTTP improves UDP-based LT (Luby Transform) code [12] for reliable delivery, which depends on FEC (Forward Error Correction) [13] with data redundancy. Since UDP cannot fairly share bandwidth with other protocols (such as TCP), TFRC (TCP Friendly Rate Control) [14] is also applied to adjust the data sending rates at servers for congestion control. The intuition behind LTTP's design is that the rate-based congestion control scheme of TFRC ensures that the sender can still send data at an appropriate rate even in face of congestion, instead of stopping sending data for a relatively long time. In addition, LT code can restore the original data without requesting for retransmission as long as the number of packet losses/errors falls into a reasonable range. Each of the two schemes is used to overcome the other's limitations: TFRC maintains reasonable bandwidth utilization, while UDP-based LT code ensures reliable data delivery. We will present the details of our design in section III, and provide analysis of the design in section IV.

We carry out NS-2 based simulations to evaluate LTTP. The results show that LTTP can maintain high goodput for many-to-one communications in different topologies, no matter what the number of servers is. On the contrary, both standard TCP and DCTCP experience goodput collapse when the number of servers is large enough. Besides, our improvement on the decoding algorithm of LT code effectively improves the goodput of LTTP, and controls the bandwidth overhead of LT code below 8%. Note that we do not suggest to completely replace TCP with LTTP in data centers for all the applications, because LTTP pays the bandwidth cost for data redundancy. However, because many-to-one communication is common in both online services and back-end computations, LTTP shows its great promise, especially when the number of servers is large.

The rest of this paper is organized as follows. Section II introduces the background knowledge. Section III describes the design of LTTP, and section IV makes the analysis. Section V presents the evaluation results. Section VI summarizes the related works. Finally Section VII concludes the paper.

## II. Background

Fig. 1 illustrates a typical setup of many-to-one communication when TCP Incast occurs. The client directly connects to a TOR (Top-Of-Rack) switch and sends data requests to a number of servers, and the servers send back the response data to the client in parallel. The many-to-one communication pattern is common in both back-end computations, e.g., GFS [3] and MapReduce [2], and online services in data centers. In GFS, files are divided into chunks and stored in different servers. Many-to-one communication occurs when a client sends reading requests to all the servers containing the chunks of a file, and the servers will reply immediately after receiving the requests. It is similar in the shuffling phase of MapReduce computation, in which each reducer receives the intermediate computation results from all the mappers before the reducing operation. As for online services, an example is that a user wants to view the photo of her new friend on the social network, and this request is injected into the data center where a front-end server redirects the request to a number of servers that each stores part of the photo.

Note that the bandwidth-delay product (BDP) in data center networks is small [8], and all the TCP connections from the servers to the client share the small BDP as well as the shallow switch buffer (typically the TOR switch directly connecting the client, as shown in Fig.1). Therefore, when all the servers send data back to the client simultaneously, it is easy that the data volume exceeds the sum of the BDP and the switch buffer size, and then packets get lost. If there are no enough Duplicated Acks to trigger fast retransmission, the servers will then wait for a time period of RTO before retransmission. Due to the mismatch between the RTT (hundreds of microseconds) and TCP's RTO (200 milliseconds), there will be a relatively long period for the TCP connection to be idle. During this idle period, the TCP connection cannot send any data, and the client cannot issue the next request, even the data from all the other servers have been received. As a result, the TCP goodput collapses.

There are many solutions to TCP Incast, DCTCP [1] and ICTCP [8] are two most recently proposed solutions. DCTCP adopts Explicit Congestion Notification (ECN) to keep the queue length in switch as short as possible, while ICTCP proposes a congestion control scheme at the receiver side, and uses receive window to throttle the incast congestion. We will discuss the related works in detail in section VI.

## III. LTTP Design

### A. Design Rationale

Since TCP's timeout is the root cause of low link utilization and goodput deterioration in TCP Incast, it naturally drives us to consider UDP instead of TCP for data transmission, because UDP has no timeout-and-retransmit mechanism. However, it is challenging to apply UDP in the many-to-one communication in data centers. First, UDP only provides best-effort service, and cannot ensure reliable data delivery. Hence, there must be an additional scheme to offer reliable data transfer, including checking data integrity and handling out-of-order delivery. Second, UDP has no congestion control. The constant sending rates at the servers will not only worsen the congestion condition when network congestion occurs, but also grab an unfair share of bandwidth against TCP flows.

We design LTTP, a UDP-based transport protocol to support many-to-one communication in data centers. LTTP employs the FEC technique to guarantee reliable packet delivery, and

applies TCP-friendly mechanism to congestion control. We choose digital fountain code [15] (specifically, LT code [12]) to achieve reliable data delivery, and adopt TFRC [14] to adjust the data sending rates and maintain reasonable bandwidth utilization. We explain the reasons that we choose these technologies to design LTTP as follows.

There is a tradeoff between bandwidth overhead (redundancy) and performance when choosing the coding scheme. For example, some erasure codes, such as Reed-Solomon erasure codes [16], have very low redundancy and can restore original data from any set of encoding data whose size is equal to that of original data; but the time that is required for encoding and decoding is unaffordable for realtime transmission, particularly considering the traffic speed in data centers is as high as 1Gbps or even 10Gbps.

We choose digital fountain code as the coding scheme. Firstly, digital fountain code can restore original data from the encoding data whose size is marginally larger than that of the original data, introducing reasonable bandwidth overhead. As we will show later in simulations, the gain we get from keeping network goodput high significantly outweighs the bandwidth cost we pay. Secondly, digital fountain code can provide good performance in encoding and decoding. Thirdly, digital fountain code only cares about how much encoding data (enough to restore original data) has been received, rather than which encoding data are received. Thus, the out-of-order delivery is not an issue any more. As a consequence, data loss caused by switch buffer overflow does not have obviously negative effect to the network throughput.

There are different implementations of digital fountain code, such as LT code [12] and Raptor code [17]. Although Raptor code outperforms LT code, we still choose LT code to realize digital fountain code in LTTP. It is because that when the data size is small, the performance difference between LT code and Raptor code is trivial [18]. In the TCP Incast scenario, the size of data exchanged between client and server is usually small. In addition, Raptor code needs to generate intermediate symbols firstly, and then uses these intermediate symbols as the input of LT code's encoding algorithm to produce the encoding data. Therefore, the implementation complexity of Raptor code is much higher than that of LT code.

As for congestion control, recent work [19] shows that when all the senders adopt digital fountain based protocols and act as selfish players to inject data in network as fast as they can, a Nash equilibrium can be reached eventually. At this equilibrium state, the throughput of each flow is similar to that when all the senders use TCP. However, in typical many-to-one communication pattern when TCP Incast occurs, the transferred data volume is very small, and it is of high probability that the Nash equilibrium cannot be reached before all the data have been transferred. So we still have to spend extra efforts to deal with congestion control in LTTP.

We simply resort to the existing TCP-friendly mechanisms, a good summary of which is presented in [20]. Generally, they can be classified into rate-based schemes and window-based schemes. To achieve TCP friendliness, the rate-based schemes adjust the sending rate based on feedback from the receiver, while the window-based schemes adopt a window (similar to the congestion window in TCP) at the sender or receiver. However, the window-based schemes may lead to a typical sawtooth pattern in the throughput. So we choose TFRC [14] as the congestion control mechanism in LTTP, which is rate-based and can provide a smoother sending rate. The analysis in section IV will show the advantage of rate-based congestion control in improving bandwidth utilization for many-to-one communication.

We emphasize that LT code can restore the original data from any set of encoding data, as long as the number of packet losses/errors falls into a reasonable range. However, in severe congestion condition, the packet losses may increase, and the receiver may not be able to restore original data within a reasonable time. In extreme cases, the receiver may fail to receive enough encoding data and restore the original data successfully. In this case, both encoding process at the sender side and decoding process at the receiver side enter an endless loop. The reason is that the encoding process does not receive the terminating signal and continues to encode data and send out encoding data, while the decoding process tries to receive more encoding data to restore the original data. This situation can be fixed by setting a timer on the sender. If the sender does not receive the terminating signal before the timer expires, the sender can terminate the communication actively.

### B. Overall Framework

The complete framework of LTTP to support many-to-one communication in data centers includes two parts, i.e., the data channel from each server to the client, and the control channel between the client and each server. In the data channel, we improve LT code for reliable data transport, and adopt TFRC for controlling the traffic sending rate at servers. The control channel is employed by the client to issue data requests to servers and send terminating signals to the servers as soon as the requested data have been restored. The servers also use the control channel to send decoding parameters to the client. The decoding parameters include the original data size and block size (the *block size* is defined at section III-C), which are used by the client to execute the decoding process (we will discuss the decoding process in section III-C). For the control channel messages, the data size is small enough to be put into a single packet. Hence, it is unnecessary to employ coding for transmission. Instead, we establish a TCP connection for each client-server pair to deliver the control channel messages reliably.

Fig. 2 illustrates the workflow in LTTP. First, the client establishes control channels (TCP connections) to all the servers. Second, the client sends requests to all the servers simultaneously through the control channel, asking the servers to start sending the data. Third, once receiving the request, the servers use control channel to send the decoding parameters back to the client. Meanwhile, each server starts to employ LT code to produce and send encoding packets continually. TFRC is used by both servers and the client to control the sending rate. Finally, as soon as the original data is successfully restored, the client sends a terminating signal through control channel back to the corresponding server, which informs the server to stop encoding.

In our implementation, the upper applications on both the server side and the client side are responsible for making

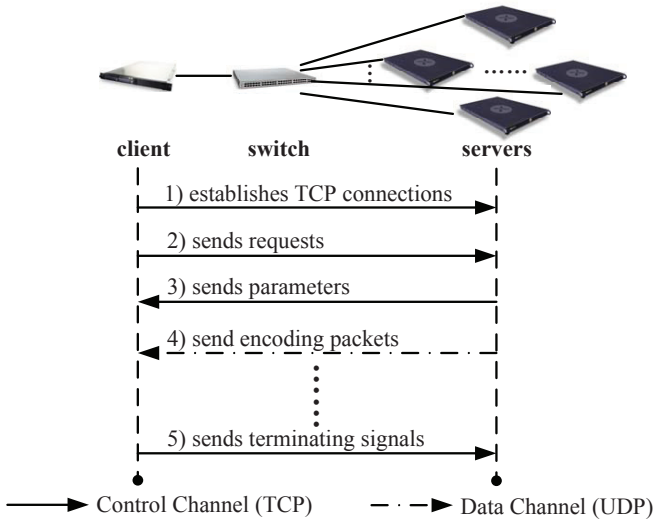



Fig. 2. The workflow of LTTP. The data channel is from servers to the client, and the control channel is between the client and servers.

decisions that when to send data and which channel to be used. For example, when the application on the server side receives a request, it calls the interface of LTTP to send decoding parameters back to the client through control channel. Next, the server starts the encoding process to generate encoding data and calls the interface of LTTP to transport the encoding data to the client side through data channel. The application on the client side calls the interface of LTTP to receive encoding data and restore the original data. Once the original data is successfully restored, the application invokes the LTTP to send the terminating signal to the server through the control channel.

Note that although we use TCP to transport the control packets in LTTP, incast congestion will not happen. Firstly, the requests and terminating signals are sent from the client to the servers. The data size is small enough to be put into a single packet. The control packets are disseminated through different switch ports to servers, so the switch buffer will not overflow. Secondly, although the decoding parameters are transmitted from the servers to the client, the data size is very small. We can use 4 bytes to store the value of both the original data size and the block size, respectively. Assume there are 45 servers. The sum of packet size is 45*(40+4+4) = 2160 bytes (the 40 bytes field is the IP header and TCP header). In the extreme case, we suppose that all the packets arrive at the switch simultaneously, and the switch port buffer is 64KB, then these packets only take up 3.29% of the buffer size.

### *C. LT Code based Transport*

LTTP depends on LT code, which is one of digital fountain codes, to realize reliable data delivery in the data channel from servers to the client.

**Digital Fountain Code:** Digital fountain [15] is a metaphor. The encoder (fountain) produces endless encoding packets (water drops), and a large number of heterogeneous decoders (buckets with different sizes) receive the encoding packets and restore the original data (buckets under the fountain to receive water drops). Once the original data has been restored, the decoder sends a terminating signal back to the encoder to inform that the data transfer has been completed.

Digital fountain code has the following features. First, it only cares about whether enough encoding data (to restore original data) has been received or not, rather than which encoding data are received. Second, for $k$ blocks of the original data, once receivers have received any $k(1+\epsilon)$ blocks of encoding data ($\epsilon$ is a small fraction, which is far less than one), the original data can be quickly restored. Third, the computation complexity of digital fountain code is almost linear in both encoding and decoding. Finally, the receiver does not need to send any feedback signal during data transmission, and the sender does not need to store and resend any data.

**LT Code Transmission:** As an implementation of digital fountain code, LT code [12] has the following encoding process. First, the original data is divided into many blocks with equal size, and each block is called an *input symbol*. The size of each input symbol is called the *block size*. Second, LT code produces *encoding symbols* on demand. Each encoding symbol is generated by employing the simple XOR (Exclusive OR) operations on distinct $d$ input symbols, where $d$ is called the *degree number* of this encoding symbol. The degree number is specified by a degree distribution called *Robust Soliton distribution* (Eq. 1). Third, LT code generates an encoding packet by combining the encoding symbol with the degree number and the indices of all the input symbols of the encoding symbol. These extra information is included in each encoding packet for decoder to restore the data. Based on the degree number, the encoding packets can be classified into two categories: single-degree encoding packets (degree number = 1) and multi-degree encoding packets (degree number $>$ 1). The encoding process is finished when the encoder receives a terminating signal from the decoder.

The decoding parameters, i.e., original data size and block size, are used in the following ways. The decoding process uses block size to decide the size of restored input symbols and assemble restored data. The original data size divided by block size equals the number of input symbols, which is used as input of the standard decoding algorithm shown in Alg. 1. The decoding process starts when a certain number of encoding packets have been received.

In Alg. 1, the decoding process restores the input symbols included in the single-degree encoding packets (lines 2-5). This set of unprocessed input symbols is called a $ripple$. Each input symbol in the $ripple$ is used to check all the received multi-degree encoding packets (lines 7-25). By comparing the encoding packets' indices and the index of input symbol, we can easily determine whether the input symbol is included in this encoding packet. If an encoding packet does not contain the input symbol, we skip this encoding packet, otherwise we perform the following three operations on the encoding packet: 1) performing XOR operations on the input symbol and the encoding symbol included in the encoding packet (line 11); 2) subtracting one from the degree number of this encoding packet (line 12); 3) updating the encoding packet's indices, namely, deleting the index of input symbol from this encoding packet's indices (line 13). After these operations, if the degree number of the encoding packet degree number changes to one,

**Algorithm 1:** Standard decoding in LT code.

```
1  StandardDecoding(set_s, set_m, input_symbol_num)
   Input: set_s: The set of single-degree encoding packets;
          set_m: The set of multi-degree encoding packets;
          input_symbol_num: The number of input symbols.
   Output: True: The decoding succeeds;
           False: The decoding fails.
2  for each packet sp ∈ set_s do
3      restore input symbol x from sp;
4      ripple.append(x)
5  end
6  restored_symbol_num ← 0;
7  while ripple ≠ NULL do
8      i ← ripple.first();
9      for each multi-degree encoding packet mp ∈ set_m do
10         if i in mp then
11             mp.encoding_symbol
               ← mp.encoding_symbol XOR i;
12             mp.degree_num ← mp.degree_num − 1;
13             update mp.indices;
14         end
15         if mp.degree_num =1 then
16             restore input symbol y from mp;
17             if y has not been restored yet then
18                 ripple.append(y);
19                 restored_symbol_num++;
20             end
21             set_m.delete(mp);
22         end
23     end
24     ripple.delete(i);
25 end
26 if restored_symbol_num == input_symbol_num then
27     return True;
28 end
29 else
30     return False;
31 end
```

**Algorithm 2:** Improved decoding of LT code.

```
1  ImprovedDecoding(set_m)
   Input: set_m: The set of multi-degree encoding packets.
   Output: NULL
2  repeat
3      repeat
4          receive an encoding packet E;
5          if E.degree_num > 1 then
6              e ← E.encoding_symbol;
7              for each input symbol d_i in e do
8                  if d_i has been restored then
9                      e ← e XOR d_i;
10                     E.degree_num ←
                       E.degree_num − 1;
11                     update E.indices;
12                 end
13             end
14         end
15         if E.degree_num = 1 then
16             restore input symbol i from E;
17             if i has not been restored yet then
18                 ripple.append(i);
19             end
20             break;
21         end
22         else
23             if E.degree_num > 1 then
24                 set_m.append(E);
25             end
26         end
27     until True;
28     while ripple ≠ NULL do
29         //This part is the same as in Alg. 1
30     end
31 until all input symbols have been restored;
```

an input symbol can be restored (line 16). If the input symbol has not been restored yet, the new input symbol expands the $ripple$ (line 18), and increases the number of restored input symbols (line 19). This encoding packet will be deleted afterwards (line 21). The input symbol will be deleted from the $ripple$ after all the received multi-degree encoding packets have been checked (line 24). The decoding process ends when the $ripple$ becomes empty.

The decoding is successful if all the input symbols have been restored (line 27), and a terminating signal will be sent back to the encoder to inform the encoder to terminate the encoding process. Otherwise it fails and the decoder continues to receive encoding packets. A newly received single-degree encoding packet will trigger another round of decoding process.

The degree distribution has a close relationship with LT code's performance. Eq. 1 shows the *Robust Soliton distribution $\mu(i)$*,

$$\mu(i) = \frac{\rho(i) + \pi(i)}{\sum_1^k \rho(i) + \pi(i)} \qquad for\ i = 1, ..., k \tag{1}$$

and $\rho$(i) and $\pi$(i) are defined as follows,

$$\rho(i) = \begin{cases} 1\ /\ k & for\ \ i{=}1 \\ 1\ /\ i(i{-}1) & for\ \ i{=}2,...,k \end{cases} \tag{2}$$

$$\tau(i) = \begin{cases} R\ /\ ik & for\ \ i\ =\ 1,\ ...,\ k\ /\ R\ -\ 1 \\ R ln(R\ /\ \delta)\ /\ k & for\ \ i\ =\ k\ /\ R \\ 0 & for\ \ i\ =\ k\ /\ R\ +\ 1,\ ...,\ k \end{cases} \tag{3}$$

where $R$ is the size of ripple and $R = c{*}\sqrt{k}{*}ln(\frac{k}{\delta})$. Parameter $k$ equals the number of input symbols, and $c$ is an appropriate constant coefficient, which is larger than zero. $\delta$ is the failure probability for the decoder to restore data. The performance of LT code can be tuned by adjusting the value of $c$ and $\delta$, which we will further study in the evaluation section.

**Improving the Decoding of LT Code:** When implementing the encoding algorithm of LT code in LTTP, we directly use

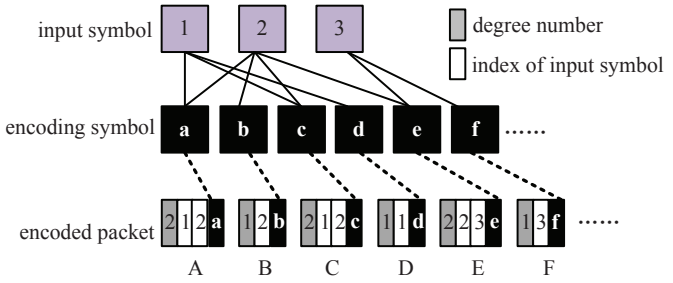



Fig. 3. An example shows the advantage of the improved decoding algorithm over the standard decoding algorithm in LT code. The standard decoding algorithm needs 6 encoding packets to successfully restore all the input symbols, while our improved decoding algorithm only requires 5.

the one suggested in [12]. But for decoding, we improve the standard algorithm in [12] for better performance, as shown in Alg. 2. We add a pre-process procedure to each received multi-degree encoding packet. The pre-process procedure firstly removes the encoding symbol from the encoding packet (line 6), and then checks all the input symbols in the encoding symbol one by one (line 7-13). If the current input symbol has been restored, we remove the input symbol from the encoding symbol by performing XOR operation (line 9), and update the degree number and indices of the encoding packet (line 10-11). After the pre-process, if the degree number of encoding packet is one, a new input symbol is restored and is added into the $ripple$ (only if the input symbol has not been restored yet) (line 18). While if the degree number of encoding packet is still larger than one, this encoding packet is added into the set of multi-degree encoding packets (line 24).

The remaining process is the same as the standard decoding process. The reasoning behind the improvement is based on the observation that, as the decoding process goes on, the number of restored input symbols increases and the probability of restoring new input symbols from the received multi-degree encoding packet also increases. Therefore, the pre-process procedure can restore some input symbols much earlier than the standard decoding algorithm, making the number of encoding packets required to restore the original data is smaller than the number needed by the standard algorithm.

We use Fig. 3 to illustrate the advantage of the improved decoding algorithm over the standard decoding algorithm in LT code. There are three input symbols with indices of 1, 2, and 3, respectively. If we use the improved decoding algorithm, we check every encoding packet when receiving it. The decoder first receives the encoding packet $A$. Because there are no restored input symbols, it does not change $A$. When the decoder receives $B$, the input symbol 2 can be restored. Then it uses the input symbol 2 to check the received encoding packet $A$, and the input symbol 1 can be restored. Next encoding packets $C$ and $D$ are received. But since the input symbols included in $C$ and $D$ have already been restored, they are not helpful to the decoding process. When packet $E$ is received, a new input symbol 3 can be restored because the input symbol 2 has been restored beforehand. So far, all the input symbols are restored.

However, if the decoder uses the standard decoding algorithm and starts the decoding process when receiving the first four encoding packets (whose size has exceeded the original data size), the decoding process will fail. It will not restart until another single-degree encoding packet (packet $F$ in this example) has been received. So the standard decoding algorithm needs 6 encoding packets to successfully restore all the input symbols, while our improved decoding algorithm only requires 5. We will carry out simulations to further evaluate the improved decoding algorithm in Section IV.

### *D. TFRC based Congestion Control*

LTTP employs TFRC [14] to conduct rate-based congestion control in the data channel. TFRC has similar phases to TCP's slow start and congestion avoidance. The server (sender) uses an initial (low) rate to start sending data and enters the slow start phase. During the slow start phase, in order to reach the fair share of bandwidth rapidly, the server doubles its sending rate in every RTT, but the increase of sending rate is no more than that of TCP, which results in that the slow start phase of TFRC takes more time than that of TCP. This is verified in our simulation (section V-E). If a feedback packet from the client (receiver) indicates that there are packet loss (i.e., loss event rate is larger than zero), the slow start phase terminates, and enters the congestion avoidance phase, in which the server uses information contained in feedback packets to measure the RTT, and puts the RTT and loss event rate into Eq. 4 to calculate the upper bound of the sending rate $T$ in bytes/sec [14],

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)} \tag{4}$$

where *s* denotes the packet size, *R* denotes the RTT, *p* denotes the steady-state loss event rate calculated by the client (which is included in the feedback packets sent back from client), and $t_{RTO}$ denotes the TCP's RTO value. Eq. 4 is a simplified version of the throughput equation from [21], which provides the detailed derivation of this equation.

The client maintains a list to record each received packet's sequence number and the timestamp. This information is feed back to the server, which uses this information to estimate the RTT. The client is also responsible for measuring the loss event rate on each data channel, which is fed back to corresponding servers by piggybacking on feedback packets through different data channels. The loss event rate is calculated as follows. If the sequence number in the list is not consecutive, packet losses are detected. A loss event includes one or more lost packets in a single RTT. The client uses the sequence number and timestamp to maintain an average loss interval, which is used to divide loss events into different loss intervals. Then the loss event rate is calculated from these loss intervals. In order to decrease the jitter of sending rate and perform only one rate update per RTT, TFRC chooses loss event rate, instead of loss rate, to notify congestion. The details of RTT measurements and calculation of loss event rate can be referred to from [22].

Furthermore, TCP uses congestion window to limit the number of on-the-fly packets. In contrast to TCP, TFRC depends on the feedback from receiver to control the data volume injected into network when congestion occurs. However, the

feedback report from receiver can be delayed by congestion or dropped by a link failure. In these cases, the sender should reduce its sending rate and even stop sending any data if these conditions continue. Hence in order to handle these situations, TFRC uses a *NoFeedBack* timer. The value of this timer roughly equals two times of RTT. When the timer is triggered, the sender halves the sending rate, and reset the timer. If this timer is expired for several times in a row, the sending rate will decrease exponentially. The sending rate will be recalculated as soon as a feedback packet is received.

## IV. Analysis

In this section, we analyze why LTTP outperforms TCP in many-to-one communications. From the discussion in section II, we know that TCP's timeout is the reason to cause performance degradation. During that period, TCP cannot send any data, resulting in bandwidth under-utilization. In contrast to TCP, LTTP adopts TFRC to implement rate-based congestion control. Let's focus on how LTTP handles congestion in many-to-one communication now.

When congestion occurs and packet gets lost, there are two possible conditions. First, no packets arrive at the receiver for a long time, so the receiver cannot measure the loss event rate, and no feedback message is sent back to the sender. If it lasts long enough that the *NoFeedBack* timer of TFRC expires, the sender will cut its sending rate in half. Second, part of the packets arrive at the receiver, but there exist sequence gaps in the received packets, which means there are packet losses. Then the receiver can calculate the loss event rate and send feedback packet to the sender. After receiving the feedback packet, the sender measures the RTT and uses Eq. 4 to calculate the new rate, and reduces its sending rate accordingly. In both conditions, LTTP does not stop sending data. Instead, it sends data at the old rate before reducing rate. Compared with TCP which stops sending data at all in the scenarios, LTTP can utilize the bandwidth much more efficiently. Although LTTP's slow response to congestion may aggravate congestion before reducing its rate, we argue that this effect is temporary, and the more efficient usage of bandwidth outweighs the negative effect.

We use NS-2 [23] to verify our analysis in a simple scenario, in which a sender and a receiver are interconnected through a switch. The sender applies LTTP to send data in one simulation while uses TCP in another. We get the behavior of LTTP and TCP in handling congestion through analyzing the trace data. Without losing generality, in order to minimize the size of trace data, we set the link bandwidth as 15 Mbps, and the link delay as 20ms (i.e., the RTT from sender to receiver is 80ms). Both simulations last for 0.6 seconds. We keep the link between the switch and the receiver down for 20ms (from 0.2s to 0.22s) to simulate a short period of severe congestion (i.e., switch buffer overflow) when many servers send data to the client in parallel in many-to-one communication. Fig. 4 shows the result.

The blue line represents the change trend of LTTP's sending rate, while the orange line shows the variation of TCP's congestion window. TCP increases its congestion window to 4 at around 0.16s, sends out 4 packets, and waits for ACKs.

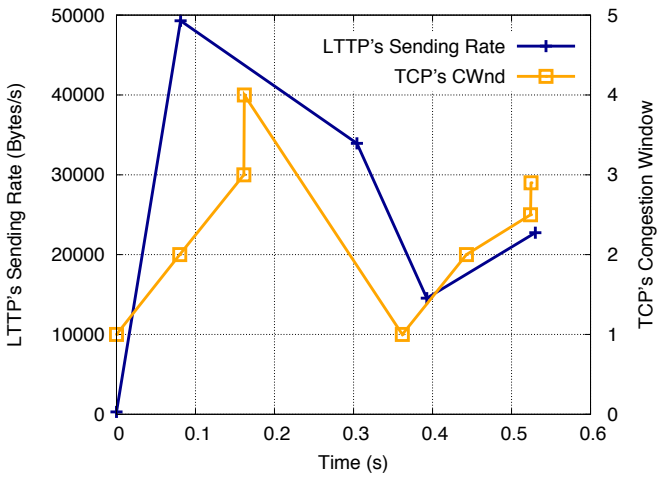



Fig. 4. TCP and LTTP's response to congestion. TCP experiences a timeout, and result in bandwidth underutilization. LTTP can efficiently utilizes the bandwidth even in face of congestion.

Unfortunately, these 4 packets are all dropped because of congestion (which lasts from 0.2s to 0.22s). The sender cannot send more data, and has to keep waiting for the ACKs until the RTO expires to reset the congestion window to 1 at about 0.36s. During this period (0.2s), TCP does not send any data, even though the bandwidth is available (from 0.22s to 0.36s). While during the congestion period, the packets that sent by LTTP are dropped either, so there is no feedback sent back to urge sender to reduce rate, and the sender still sends packets at the old rate. However, once the congestion situation mitigates and the bandwidth is available, packets will successfully arrive at the receiver. Then the receiver will calculate loss event rate and send feedback packet to ask the sender to reduce its rate. We can verify this handling from the result. After the congestion, the sender firstly adjusts its rate at about 0.3s. It is roughly one RTT (0.08s) after congestion disappears (0.3 - 0.22 = 0.08s), which is exactly the period from sending a packet to receiving its feedback packet. The result also demonstrates that the change trend of LTTP's sending rate is similar to that of TCP's congestion window.

## V. Evaluation

### A. Simulation Setup

We use NS-2 [23] to evaluate the performance of LTTP. We let the client and all the servers connect to a single switch, and the client and all the servers deploy LTTP. All links have 1Gpbs capacity, and the RTT is set as 100 microseconds. We simulate the many-to-one communication by letting the client request data from various numbers of servers. We observe the goodput at the client. If not specified, each switch port has exclusively 64KB buffer. We vary the data chunk size from 64KB, 128KB, to 256KB. For all simulations, the block size in LT code is 1024 bytes. Note that the XOR operations in the encoding and decoding process of LT code only introduce linear computation overhead, hence in the simulation we do not take the processing delay in encoding/decoding into consideration when measuring the transmission delay.

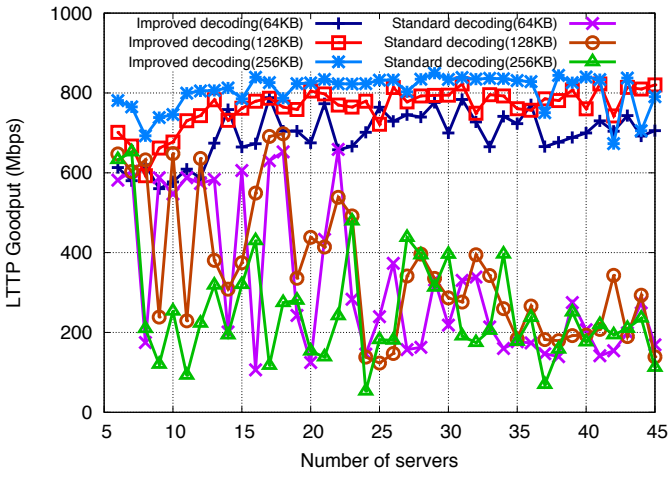



Fig. 5. Goodput of LTTP with different decoding algorithms. The improved decoding algorithm greatly promotes the LTTP goodput.

### B. Parameters in LT Code

As stated in Section III, we adjust the value of constant coefficient $c$ and failure probability $\delta$ when calculating $R$ in Eq. 3 to tune the performance of LT code. We firstly evaluate the impacts of the two parameters on the goodput of LTTP. We vary $c$ within [0.1,10] and $\delta$ within [0.1,0.9]. Because we only care the performance of LT code here, we use just one client-server pair in this group of simulation.

From the results, we find that the failure probability $\delta$ has no obvious impact on the LTTP goodput, and the LTTP goodput is higher when the value of constant coefficient $c$ is less than 1. Due to space limitations, we omit the details of simulation results. For all the following simulations, we choose $c = 0.2$ and $\delta = 0.7$.

### C. Decoding Improvement of LT Code

Then we conduct simulations to evaluate our improvement on the decoding algorithm of LT code. We use both the standard LT decoding algorithm and the improved LT decoding algorithm in LTTP to compare their goodputs.

From the simulation result shown in Fig. 5, we can see that the improved decoding algorithm greatly promotes the goodput of LTTP, especially when the number of servers is large. For instance, when there are 45 servers and the data size is 256KB, the goodput of the standard LT code is 112.18Mbps while that of improved LT code is 790.48Mbps. The enhancement comes from that the improved decoding algorithm needs less encoding packets to restore the original data, which will reduce the amount of transferred data in the network, and accordingly improves the network goodput. Besides, we find that larger data size results in higher throughput in LTTP. It is because larger data size can produce *ripple* with larger size, which means during the decoding process, there are more input symbols used to restore other input symbols. For all the following simulations, we use the improved decoding algorithm in LTTP.

TABLE I
BANDWIDTH OVERHEAD OF LTTP

| Metric | *Data Size* | | |
|---|---|---|---|
| | ***64KB*** | ***128KB*** | ***256KB*** |
| Number of received packets | 68 | 137 | 271 |
| Average degree number | 4.29 | 4.73 | 6.88 |
| Bandwidth overhead (%) | 6.62 | 7.29 | 6.27 |

### D. Bandwidth Overhead of LTTP

We define the *bandwidth overhead* of LTTP as the ratio of additional data size introduced by LT code over the original data size. The bandwidth overhead comes from two parts. First, LT code needs extra bytes to record the encoding information for receiver to successfully restore the original data. In our current implementation, for each encoding packet, we use one byte to record the degree number and one byte for the index of each input symbol. Second, LT code introduces redundancy, so the size of encoding symbols required to successfully restore the original data is larger than the original data size.

We run the simulation on a single pair of client and server, for data sizes of 64KB, 128KB and 256KB, respectively. The block size is still set as 1024 bytes. During the simulation period, the encoding data received at the client are recorded, and then we use the information included in these encoding data to calculate the bandwidth overhead.

The results are shown in Table I. Generally, the bandwidth overhead of LTTP is between 6-8%, which is affordable in most cases. We can further adopt other schemes to reduce the bandwidth overhead, such as arithmetic coding [24], which can encode the indices of all the input symbols in each encoding symbol as a single double number. However, the cost is the increased computation overhead at the servers and the client.

### E. LTTP Vs. TCP

We now compare LTTP with TCP to check the effectiveness of LTTP in mitigating TCP Incast. We run both LTTP and TCP NewReno in the many-to-one communication scenario. The number of servers is varied from 1 to 45. For TCP, we choose the packet size of 1024 bytes, which equals the block size adopted in LT code. In order to mitigate the impact of randomness of LT code, we run every simulation of LTTP for 10 times, and calculate the average goodput. Fig. 6 shows the results.

We find from the figure that when the number of servers is small (less than 15), TCP Incast hardly occurs. Under this condition, TCP outperforms LTTP, because LT code introduces bandwidth overhead (as measured in Section V-D). In our simulation, the goodput of TCP is about 950 Mbps, while that of LTTP is around 700Mbps. But as the number of servers increases, the data sent from the servers easily overflows the switch buffer, which causes TCP Incast and goodput degradation as low as tens of Mbps. However, the goodput of LTTP maintains above 750Mbps in most settings. In the case when the number of servers is 36 and the data size is

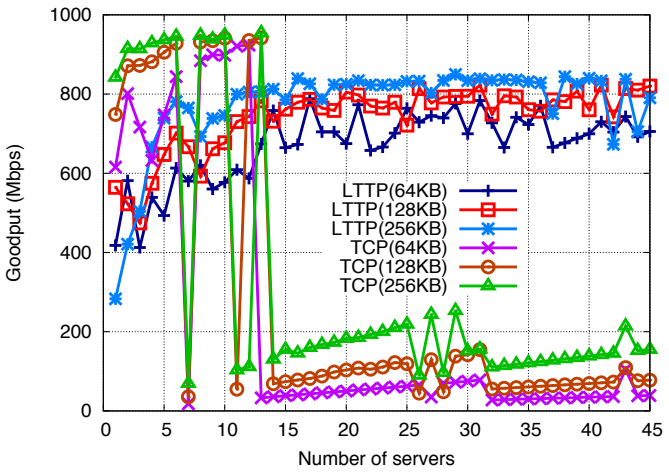



Fig. 6. Goodput of TCP and LTTP against the number of servers in many-to-one communication. LTTP never experiences goodput collapse.

64KB, the goodput of LTTP is 770Mbps while that of TCP is only 93Mbps.

The reason that LTTP achieves much higher goodput than TCP when the number of servers is large lies in two aspects. On one hand, LT code only cares whether enough encoding packets have been received or not, instead of which encoding packets have been received. So even when switch buffer experiences overflow (this cannot be avoided, because in extreme cases, only two packets from each server can overflow the switch buffer), the dropped data will not bring many negative effects to LTTP. On the other hand, servers do not adopt the timeout-and-retransmit mechanism, so the packet loss caused by buffer overflow does not cause the servers to stop sending data over a relatively long time.

We also observe that when the number of servers is smaller than 5, the goodput of LTTP is less than 600Mbps. The reason is that TFRC is less aggressive than TCP to grab available bandwidth in slow start phase, and increases its sending rate more slowly than TCP [14]. Hence, the slow start phase of TFRC takes more time than that of TCP. Meanwhile, when the number of servers is small, there is no packet loss. Considering the data volume sent by every server is small, the time spent on the slow start phase accounts for the majority of the whole transfer time. Since the slow start phase of TFRC takes more time than that of TCP, LTTP obtains lower goodput compared with TCP. We should note that, ICTCP [8] also encounters this phenomenon. However, from the simulation results, we can clearly see that the low performance of LTTP during the slow start phase does not affect its performance when the number of servers increases. When the number of servers is large, the LTTP goodput (700∼800Mbps) is less than the theoretical value of ≈900Mbps (considering about 8% bandwidth overhead of LTTP). This is attributed to the fact that when the number of servers is large, there will be packet loss when multiple servers send the data simultaneously, which may cut down the goodput.

### *F. LTTP Vs. DCTCP*

Next we compare LTTP with DCTCP, which mitigates TCP Incast by controlling the utilization of switch buffers. In DCTCP, the low and high thresholds of ECN are set as the same value, which is specified by *marking threshold K*. DCTCP detects the congestion condition by estimating the fraction of marked packets, and accordingly decreases the congestion window properly. In estimating the fraction of marked packets, DCTCP uses a parameter $g$ $(0 < g < 1)$ to represent the weight given to new samples, so as to smooth the traffic sending rate. We use the Stanford implementation on NS-2.35 [25] to run DCTCP. Similar as the simulations above, we choose packet size of 1024 bytes and vary the number of servers from 6 to 45. We also run every simulation of LTTP for 10 times to mitigate the impact of randomness of LT code, and calculate the average goodput. In this group simulations, we set two different buffer sizes for each switch port, i.e., 64KB and 128KB.

**Parameter Choice in DCTCP:** We conduct a group of simulations to figure out the optimal parameters in DCTCP, particularly, the marking threshold $K$ and the sampling weight $g$, for different switch buffer sizes. The results show that when the switch buffer size is 64KB per port, the optimal parameter values are $K = 4$ and $g = 0.15$; while when the switch buffer size per port is 128KB, the optimal parameter values are $K = 20$ and $g = 0.0625$, which verifies the result in [1].

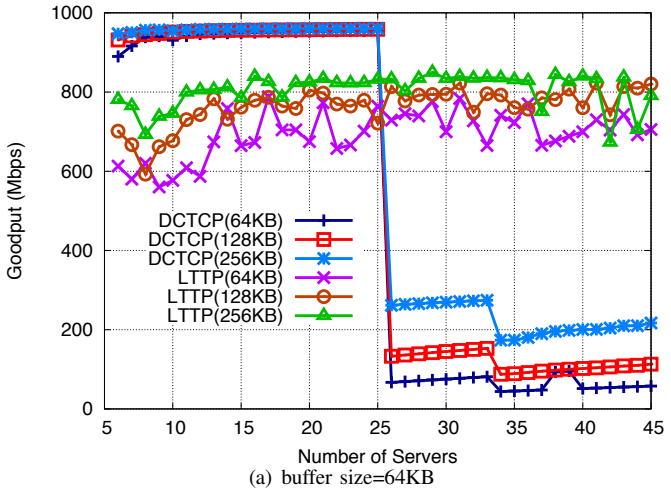
**Goodput Comparison between LTTP and DCTCP:** We compare the goodput between DCTCP and LTTP with different data sizes fetched from servers, i.e., 64KB, 128KB, and 256KB. Fig. 7 shows the results.

We find that the goodput of LTTP does not vary much with different numbers of servers, and keeps the goodput around 700Mbps (for larger data size, the goodput even exceeds 800Mbps). However, the goodput of DCTCP depends on the number of servers in the many-to-one communication. When the number of servers is small, the DCTCP goodput is as high as 950Mbps. But when the number of servers is more than 25 for 64KB switch buffer and more than 34 for 128KB switch buffer, DCTCP experiences goodput collapse. It is because when the number of servers is small, even the data from all the servers arrive at the output port of the switch simultaneously, the buffer size is still enough to hold them. But when the number of servers is large enough, even if each server sends a single packet, buffer overflow and TCP Incast can occur.

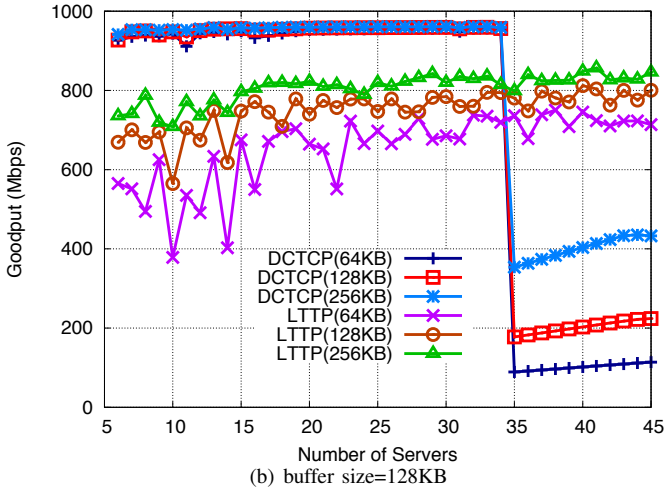
Putting Fig. 7 and Fig. 6 together, we get that compared with standard TCP, DCTCP actually increases the number of servers it can support before TCP Incast occurs, instead of fundamentally solve the TCP Incast problem. However, the goodput of LTTP does not degrade with the growth of the number of servers, and the phenomenon of goodput collapse never happens. Therefore, LTTP shows its great promise in supporting many-to-one communication in data centers, especially when the number of servers is large.

**Congestion Window Evolvement in DCTCP:** We check the evolvement of the congestion window (CWND) on a server in DCTCP to clearly explain why DCTCP still experiences TCP Incast when the number of servers is large. We use 16 servers for parallel data transmission to one client. The data volume sent by each server is 64KB. The buffer size of the switch port is 64KB, and the packet size is 1024 bytes.

The result is shown in Fig. 8. We can see that before timeout occurs, the size of CWND is 2.929. The number of in-fly




(a) buffer size=64KB




(b) buffer size=128KB

Fig. 7. Goodput of DCTCP and LTTP against the number of servers in many-to-one communication. DCTCP also experiences goodput collapse when the number of servers is large enough.

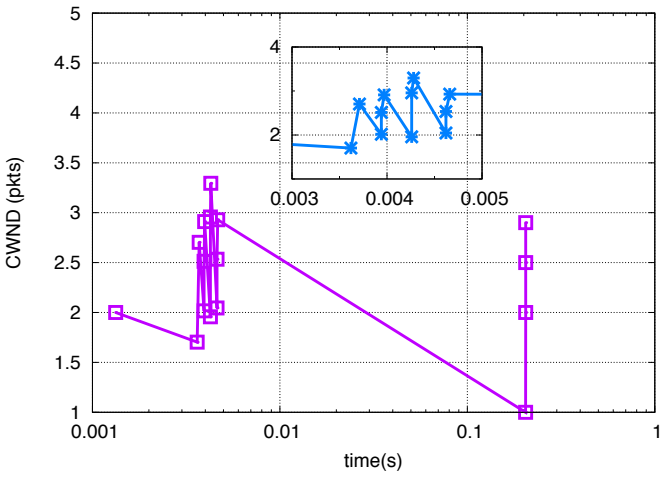



Fig. 8. The evolvement of CWND at a server which experiences a timeout in DCTCP. No enough ACKs to trigger retransmission.

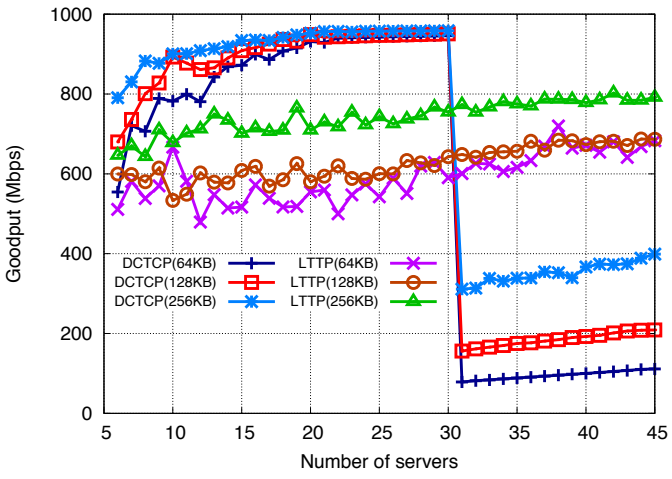


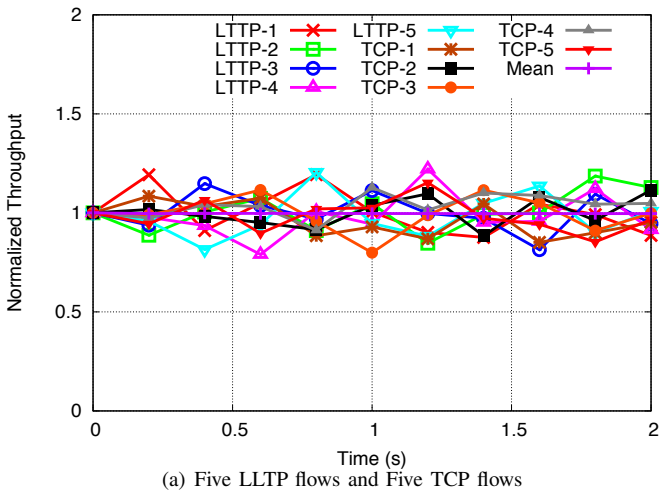

Fig. 9. Goodput of DCTCP and LTTP against the number of servers in many-to-one communication under Fat-Tree topology. Both DCTCP and LTTP showcase the same trend as previous simulations. Compared with previous simulations, the goodput is reduced slightly, which is caused by the larger RTT.

packets sent by the server is thus 2. Then at $t = 0.00466s$, the first packet gets dropped, while the second packet is successfully delivered to the client. So the client sends a duplicate ACK back to the server. However, one duplicate ACK is not enough to trigger fast retransmission at the server, and the server cannot send any new data until the retransmission timer expires.

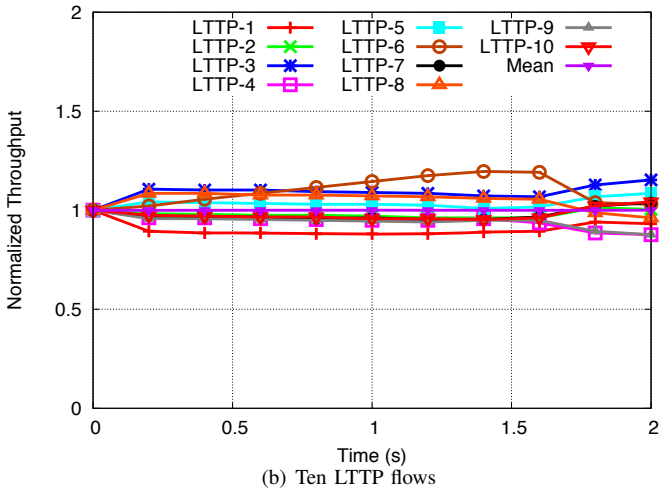
### *G. Fat-Tree Topology*

Previous simulations are conducted under a simple topology, i.e., all servers and the client are interconnected by a single switch. In order to verify the performance of LTTP under different network topologies, we compare LTTP with DCTCP under Fat-Tree network. In this simulation, we use an 8-ary Fat-Tree topology, which includes 128 hosts in total. We set the bandwidth of each link as 1 Gbps, and the delay for each link is 25 microseconds. The switch port buffer is set to be 128KB. As previous simulations, we set both the packet size of TCP and the block size of LTTP to be 1024 bytes. Without loss of generality, when simulating the many-to-one communication pattern, we select the left-most host as the client, and randomly select a different number of servers among other hosts. We vary the number of servers from 6 to 45.

Fig. 9 illustrates the performance comparison between LTTP and DCTCP under Fat-Tree topology. From the result, we can see that both LTTP and DCTCP achieve slightly reduced goodput compared with that in Fig. 7(b). This can be attributed to the increase of RTT between servers and the client. We should note that in previous simulations, the paths between servers and the client have the same length (i.e., 2 hops). Hence the RTTs between servers and the client are equal (100 microseconds). While in Fat-Tree topology, the servers locate in different pods, so the paths between the client and servers have different hops. Accordingly, the RTTs between different servers and the client also have different values, i.e., 100 microseconds (2 hops), 200 microseconds (4 hops), and 300 microseconds (6 hops), respectively. Both LTTP and DCTCP in Fig. 9 and Fig.7(b) clearly showcase the same trend: DCTCP outperforms LTTP when the number




(a) Five LLTP flows and Five TCP flows




(b) Ten LTTP flows

Fig. 10. Normalized throughput is calculated every 200ms when different servers continuously send data to a same client under the same switch.

of servers is small. However, DCTCP experiences goodput deterioration when selecting more servers to participate in the many-to-one communication, and LTTP still maintains goodput around 700Mbps throughout the simulation. The reason behind is the same as the one we discussed previously in section V-F. We also compare LTTP with TCP under Fat-Tree topology. The simulation result is similar to that shown in Fig. 6, so we omit the simulation results.

### *H. Fairness of LTTP*

We conduct two simulations to evaluate the fairness of LTTP. The first simulation includes 5 TCP flows and 5 LTTP flows, and the second simulation uses 10 LTTP flows. In the two simulations, all flows continuously send data from different servers to the same client under the same switch. We define *normalized throughput* as the ratio of measured flow throughput over that of the expected. If the *normalized throughput* is closer to 1, the fairness among flows is better. Fig. 10 shows the results, which tells that LTTP flows can fairly share bandwidth with both other LTTP flows and TCP flows. Besides, the overall utilization of the bottleneck link is always above 99%, indicating that LTTP can make efficient utilization of link bandwidth, even together with TCP.

## VI. Related Works

Nagle et al. first identify and describe the TCP Incast phenomenon in distributed storage clusters [6]. By analyzing the simulation traces, Zhang et al. find that the TCP throughput deterioration is primarily attributed to two types of timeouts, namely, BHTO (Block Head TimeOut) and BTTO (Block Tail TimeOut) [26]. When the number of concurrent servers is small, BTTO is the main cause for goodput drops. It means that at least one of the last three packets is lost, and thus TCP cannot trigger fast retransmission, which results in timeout. On the other side, when the number of servers is large, BHTO dominates the goodput degradation. It occurs when the first window of the data blocks are totally lost, and no more data can be sent from the servers until the retransmission timer expires.

The fundamental reason for goodput collapse in TCP Incast can be attributed to three factors [7]: 1) barrier-synchronized workload, 2) mismatch between TCP's RTO and RTT in data center networks, and 3) limited buffer size in switches. Since barrier-synchronized workload is determined by applications with many-to-one communication pattern, all the previous solutions focus on the latter two factors, i.e., either reducing the RTO or avoiding switch buffer overflow.

The first category of solutions mitigate TCP Incast by setting the RTO value to microsecond granularity [7], [11], which matches the RTT in data centers. Though this type of solutions are effective in mitigating TCP Incast, updates to OS and hardware are required to implement microsecond-grained timers. Furthermore, with the introduction of optical fibers and optical switches in data centers, even microsecond-grained RTO may not be able to solve the problem [26].

The second category of solutions modify QCN (Quantized Congestion Notification), a Layer 2 end-to-end congestion management scheme introduced in IEEE 802.1Qau [27], to smartly control utilization of the switch buffer. In essence, the framework of QCN includes two parts: 1) QCN-enabled switch. The switch is responsible for monitoring the presence of congestion in the network and notifying it to the source. 2) QCN-enabled Network Interface Card (NIC), which receives the congestion notification sent from switches, and decreases the sending rate to alleviate the congestion. However, QCN gets poor performance in TCP Incast setting [9]. The reason is that QCN cannot ensure rate fairness among different flows which share the same bottleneck link. FQCN [9] modifies the algorithm in QCN-enabled switches to achieve fairness among flows sharing the same bottleneck link, and mitigates the TCP Incast. AF-QCN [10] proposes an algorithm used by the switches to achieve flexible weighted fair share of bandwidth for flows. However, these solutions need to update both the end hosts and switches.

In addition, the third category of solutions try to keep the switch buffer utilization low and avoid timeouts through modifying the congestion control algorithm of TCP, including ICTCP [8] and DCTCP [1]. ICTCP modifies the congestion control algorithm by adjusting the receive window size of each TCP connection based on the measured available bandwidth at the receiver, which further controls the sending rates of the senders and prevents the switch buffer from over-

TABLE II
COMPARISON BETWEEN DIFFERENT SOLUTIONS

| Metrics | RTO ($us$) | DCTCP | ICTCP | FQCN/ AF-QCN | LTTP |
|---|---|---|---|---|---|
| OS update on end host | ✓ | ✓ | ✓ | × | ✓ |
| Hardware update on end host | ✓ | × | × | ✓ | × |
| Switch support | × | ✓ | × | ✓ | × |
| Performance degradation with more servers | ✓ | ✓ | ✓ | × | × |

flow. DCTCP's core idea is to leverage Explicit Congestion Notification (ECN) to keep the queue length in switch as short as possible, while ensuring high network throughput. DCTCP also modifies ECN by setting both the low and high threshold of ECN as the same value. A major difference between DCTCP and TCP is that, DCTCP does not always cut the congestion window in half when ECN notifications are received; instead, it decreases the congestion window properly according to the fraction of marked packets. Nevertheless, ICTCP and DCTCP cannot fundamentally avoid TCP Incast, because bursty traffic can indeed overflow the switch buffer. As shown in our NS-2 based simulations (Section V-F), when the number of servers is large and there exists traffic burst, packet loss can still occur, which results in goodput degradation.

In [28], we present some preliminary results of applying coding to prevent TCP Incast. But in this paper, we make improvements on the decoding algorithm of LT code and design LTTP as a transport library for many-to-one communications in data centers.

We compare the existing solutions with LTTP in Table II. We get that implementing microsecond-grained RTO requires updating OS and hardware, DCTCP needs ECN-enabled switches to achieve congestion control. Similar to LTTP, ICTCP only needs to update OS on end hosts, but it cannot ensure good performance when the number of servers is large ( [8] shows that ICTCP experiences timeout in up to 6% of all experiments). Only LTTP and FQCN/AF-QCN can maintain good performance under different conditions, while FQCN/AF-QCN also needs to update both the end hosts' hardware and switches.

## VII. CONCLUSION

TCP Incast in data center networks causes goodput collapse. Existing approaches either require updating the OS/hardware, or cannot fundamentally solve the problem when the number of servers is large enough. In this paper we design LTTP, a novel transport protocol for many-to-one communications in data center networks. LTTP improves the LT code for data transmission from servers to the client, and adopts TFRC to control the traffic sending rates at servers. Though LTTP bears bandwidth overhead due to the data redundancy in coding, it does not experience goodput degradation with the growth of the number of servers. NS-2 based simulations show that LTTP can effectively avoid goodput collapse and significantly outperforms DCTCP when the number of servers is large. Simulation results also demonstrate that LTTP has good fairness under different settings.

## ACKNOWLEDGMENTS

The work was supported by the National Basic Research Program of China (973 program) under Grant 2014CB347800, 2012CB315803, the National High-tech R&D Program of China (863 program) under Grant 2013AA013303, and the Natural Science Foundation of China under Grant No.61170291, No.61133006, No.61161140454.

## REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM 2010*, New York, NY, USA, 2010, pp. 63–74.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. OSDI'04*, Berkeley, CA, USA, 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. SOSP '03*, New York, NY, USA, 2003, pp. 29–43.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM 2008*, New York, NY, USA, 2008, pp. 63–74.

[5] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proc. ACM SIGCOMM 2009*, New York, NY, USA, 2009, pp. 51–62.

[6] D. Nagle, D. Serenyi, and A. Matthews, "The panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage," in *Proc. 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004, pp. 53–62.

[7] 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 *Proc. SIGCOMM '09*, 2009, pp. 303–314.

[8] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: incast congestion control for TCP in data center networks," in *Proc. Co-NEXT '10*, New York, NY, USA, 2010.

[9] Z. Yan and N. Ansari, "On mitigating tcp incast in data center networks," in *Proc. IEEE INFOCOM 2011*, Apr. 2011, pp. 51–55.

[10] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers," in *IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2010, pp. 58–65.

[11] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. WREN '09*, New York, NY, USA, 2009, pp. 73–82.

[12] M. Luby, "LT codes," in *Proc. 43rd Symposium on Foundations of Computer Science*, 2002, pp. 271 – 280.

[13] A. J. McAuley, "Reliable broadband communication using a burst erasure correcting code," in *Proc. SIGCOMM '90*, 1990, pp. 297–306.

[14] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *Proc. SIGCOMM '00*, New York, NY, USA, 2000, pp. 43–56.

[15] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM '98*, New York, NY, USA, 1998, pp. 56–67.

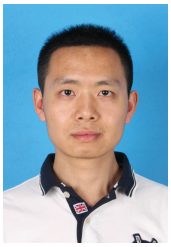
[16] I. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[17] A. Shokrollahi, "Raptor codes," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

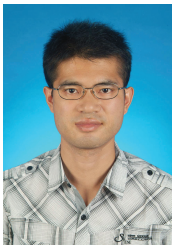
[18] P. Cataldi, M. Shatarski, M. Grangetto, and E. Magli, "Implementation and Performance Evaluation of LT and Raptor Codes for Multimedia Applications," in *Intelligent Information Hiding and Multimedia Signal Processing, IIH-MSP '06*, 2006, pp. 263–266.

[19] L. Lpez, A. Fernndez, and V. Cholvi, "A game theoretic comparison of TCP and digital fountain based protocols," *Comput. Netw.*, vol. 51, no. 12, pp. 3413–3426, 2007.

[20] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-friendly congestion control," *IEEE Network*, vol. 15, no. 3, pp. 28 –37, 2001.
[21] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling tcp throughput: a simple model and its empirical validation," in *Proc. ACM SIGCOMM '98*, New York, NY, USA, 1998, pp. 303–314.
[22] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification." [Online]. Available: http://www.ietf.org/rfc/rfc5348.txt
[23] The Network Simulator - NS 2. [Online]. Available: http://isi.edu/nsnam/ns/
[24] Arithmetic coding. [Online]. Available: http://en.wikipedia.org/wiki/Arithmetic_Coding
[25] DCTCP implementation for ns-2.35. [Online]. Available: http://www.stanford.edu/~alizade/Site/DCTCP_files/dctcp-ns2-rev1.0.tar.gz
[26] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding TCP incast in data center networks," in *Proc. IEEE INFOCOM 2011*, Apr. 2011, pp. 1377 –1385.
[27] IEEE 802.1Qau. [Online]. Available: http://www.ieee802.org/1/pages/802.1au.html
[28] J. Changlin, L. Dan, X. Mingwei, and Z. Kai, "A coding-based approach to mitigate TCP Incast in data center networks," in *Proc. the ICDCS Workshop on data center performance*, 2012.



**Changlin Jiang** received the B.S. and M.S. degrees from the Institute of Communication Engineering, PLA University of Science and Techology in 2001 and 2004 respectively. Now, he is a Ph.D. candidate in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include Internet architecture, data center network, network routing.



**Dan Li** received the M.E. degree and Ph.D from Tsinghua University in 2005 and 2007 respectively, both in computer science. Before that, he spent four undergraduate years in Beijing Normal University and got a B.S. degree in 2003, also in computer science. He joined Microsoft Research Asia in Jan 2008, where he worked as an associate researcher in Wireless and Networking Group until Feb 2010. He joined the faculty of Tsinghua University in Mar 2010, where he is now an Associate Professor in Computer Science Department. His research interests include Internet architecture and protocol design, data center network, software defined networking.

**Mingwei Xu** received the B.S. degree in 1994 and Ph.D. degree in 1998 both from the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He is currently a professor in Tsinghua University. His research interests include computer network architecture, Internet protocol and routing, high-speed router architecture and green networking.