Providing Reliable Service in Data-center Networks

A.Suresh[1], S. Jaya Kumar [2]
[1]M.Tech (CSE) Student, Department of CSE, SRM University, Ramapuram, Chennai, India
suresh_hce2004@yahoo.co.in
[2]Assistant Professor, Department of CSE, SRM University, Ramapuram, Chennai, India, ven_bsc@yahoo.co.in

Abstract
Transport Control Protocol (TCP) in cast congestion happens in high-bandwidth and low-latency networks when multiple synchronized servers send data to the same receiver in parallel. For many important data-center applications such as Map Reduce and Search, this many-to- one traffic pattern is common. Hence TCP in cast congestion may severely degrade their performances, e.g., by increasing response time. In this paper, we study TCP in cast in detail by focusing on the relationships between TCP throughputs, round-trip time (RTT), and receive window. Unlike previous approaches, which mitigate the impact of TCP in cast congestion by using a fine- grained timeout value, our idea is to design an In cast congestion Control for TCP (ICTCP) scheme on the receiver side. In particular, our method adjusts the TCP receive window proactively before packet loss occurs. The implementation and experiments in our test bed demonstrate that we achieve almost zero timeouts and high good put for TCP in cast. Hence, TCP connections with short round-trip times may receive unfairly large allocations of network bandwidth when compared to connections with longer round-trip times.

Keywords: Distributed Network Management, Data Center Network, TCP Incast Congestion

1. Introduction
This paper focuses on avoiding packet loss before in cast congestion, which is more appealing than recovery after loss. Of course, recovery schemes can be complementary to congestion avoidance.
The smaller the change we make to the existing system, the better. To this end, a solution that modifies only the TCP receiver is preferred over solutions that require switch and router support (such as ECN) and modifications on both the TCP sender and receiver sides. Our idea is to perform in cast congestion avoidance at the receiver side by preventing in cast congestion. The receiver side is a natural choice since it knows the throughput of all TCP connections and the available bandwidth. The receiver side can ad-just the receive window size of each TCP connection, so the aggregate burstiness of all the synchronized senders are kept under control. We call our design in cast congestion Control for TCP (ICTCP).
TCP does not work well for many-to -one traffic pat-terns on high-bandwidth, low-latency networks. Congestion occurs when many synchronized servers under the same Gigabit Ethernet switch simultaneously send data to one receiver in parallel. Only after all connections have finished the data transmission can the next round be issued. Thus, these connections are also called barrier-synchronized. The final performance is determined by the slowest TCP connection, which may suffer from timeout due to packet loss. The performance collapse of these many-to-one TCP connections is called TCP in cast congestion. The root cause of TCP in cast collapse is that the highly burst traffic of multiple TCP connections overflows the Ethernet switch buffer in a short period of time, causing intense packet loss and thus TCP retransmission and timeouts. Previous solutions focused on either reducing the wait time for packet loss recovery with faster retransmissions [2], or controlling switch buffer occupation to avoid overflow by using ECN and modified TCP on both the sender and receiver sides [5].
However, adequately controlling the receive window is challenging: The receive window should be small enough to avoid in cast congestion, but also large enough for good performance and other non in cast cases. A well-performing throttling rate for one in cast scenario may not be a good fit for other scenarios due to the dynamics of the number of connections, traffic volume, network conditions, etc.

2. Problem statement

The existing algorithm focuses on the issue of estimating the used bandwidth by counting ACK packets and by filtering the information they convey. The ability of this method to perform well when the data is not perfect is crucial for the extension of the method from spheres (for which it was initially developed) to arbitrary shapes [2].The Proposed algorithm redirects the packet to header parser module. Packet header is parsed and the information on flow table is updated. Algorithm module is responsible for receive window calculation .If a TCP ACK packet is sent out, the header modifier change the receive window field in TCP header if need. Client server communication acknowledgement is available .It is a variable bandwidth based on packet size. Packet re-transmission can be done by using intermediate router. With the support of TOR switch we can make connection with dozens of servers [4].Amount of data transmitted by each connection relatively small. The files are deliberately stored in multiple servers. Packet loss is controlled by using ICTCP algorithm [5].

2.1 Bandwidth allocation for Data Center

Running multiple virtual networks over a real physical substrate is a promising way to provide agility in current data centers.[2] However, such virtual networks may experience severely degraded performance due to the competing of network traffic on shared physical links. Based on the idea of the Stack solution from non-cooperative game theory, this paper presents a hierarchical game theoretic model for dynamic bandwidth allocation between virtual networks, which can be stable and can maximize the revenue of both infrastructure providers who manage the physical infrastructure and service providers who utilize the virtual networks to provide services. In the model, the data center owner as a leader designs a pricing mechanism for bandwidth allocation that attempts to drive the virtual networks to the social optimal solution, each virtual network as a follower chooses a willingness-to-pay to maximize its own profit. Experimental results show that the bandwidth allocation between virtual networks is efficient and fair [6].

The traffic and network conditions in data-center networks create the three preconditions for in cast congestion as summarized in [2]. First, data-center networks are well structured and layered to achieve high bandwidth and low latency, and the buffer size of top - of-rack (ToR) Ethernet switches is usually small. Second, a recent measurement.

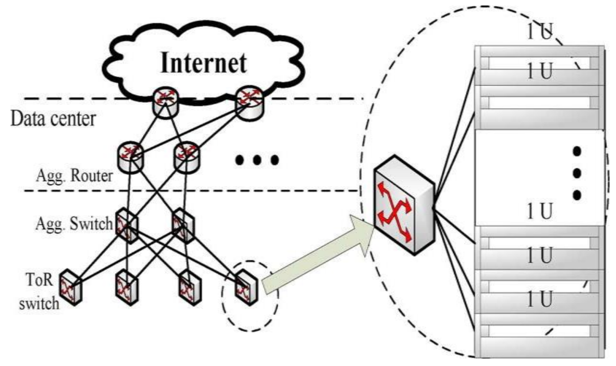


Fig. 1.Data-Center Network

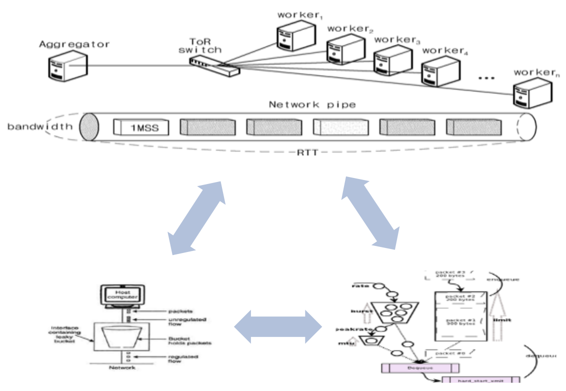


Figure2. System Architecture Diagram

Once the system has been designed, the next step is to convert the designed one in to actual code, so as to satisfy the user requirements as excepted. If the system is approved to be error free it can be implemented. When the initial design was done for the system, the department was consulted for acceptance of the design so that further proceedings of the system development can be carried on. After the development of the system, a demonstration was given to them about working of the system. The aim of the system illustration was to identify any malfunctioning of the system.

2.2 Citations and References

[1].S.Kandula,S.Sengupta, A.Greenberg,P.Patel,and R.Chaiken,"The nature of data center traffic: Measurements & analysis,"inProc.IMC,2009-We explore the nature of traffic in data centers, designed to sup-port the mining of massive data sets. We instrument the servers to collect socket-level logs, with negligible performance impact. In a 1500 server operational cluster, we thus amass roughly a terabyte of measurements over two months, from which we obtain and report detailed views of traffic and congestion conditions and patterns. We further consider whether traffic matrices in the cluster might be obtained instead via tomography inference from coarser-grained counter data.
[2].Ganger,G.Gibson,and B.Mueller,"Safe and effective fine-grained TCP retransmissions for datacenter communication," inProc.ACM SIGCOMM, 2009-This paper presents a practical solution to a problem facing high-fan-in, high-bandwidth synchronized TCP workloads in datacenter Ethernets the TCP in cast problem.
[3]. S.Ghemawat,"MapReduce: Simplified data processing on large clusters," in Proc. OSDI, 2004, -Map Reduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the pro-gram's execution across a set of machines, handling ma-chine failures, and managing the required inter-machine communication.

3. ICTCP Algorithm

ICTCP provides a receive window- based congestion control algorithm for TCP at the end-system. The receive windows of all low-RTT TCP connections are jointly adjusted to control throughput on incast Congestion [7]. Our ICTCP algorithm closely follows the design points made in Section III. In this section, we describe how to set the receive window of a TCP connection [8].There are several benefits that can be achieved when ICTCP is implemented in a driver:[2] 1) It naturally supports virtual machines, which are widely used in data centers. We discuss this point in detail in the following section. 2) ICTCP needs the incoming throughput on a very small time granularity (comparable to RTT at hundreds of microseconds) to estimate available bandwidth, and this information can be easily obtained at a driver. Note that the incoming traffic includes all types of traffic arriving on that interface, besides TCP. 3) It does not touch TCP/IP implementation in the Windows kernel. As a quick and dirty solution, it supports all OS versions instead of patching each one by one for deployment in a large data-center network with various TCP implementations.

3.1 ICTCP pseudo-code

```
Initially:
cwnd = 1;
ssthresh = infinite;
New ack received:
if (cwnd < ssthresh)
/* Slow Start*/
cwnd = cwnd + 1;
else
/* Congestion Avoidance */
cwnd = cwnd + 1/cwnd;
Timeout:
/* Multiplicative decrease */
ssthresh = cwnd/2;
cwnd = 1;
```

Sliding windows, a technique also known as windowing, is used by the Internet's Transmission Control Protocol (TCP) as a method of controlling the flow of packets between two computers or network hosts [3]. TCP requires that all transmitted data be acknowledged by the receiving host. Sliding windows is a method by which multiple packets of data can be affirmed with a single acknowledgment**.** DCTCP is an enhancement to the TCP congestion control algorithm for data center networks. It leverages Explicit Congestion Notification (ECN)[1], a feature which is increasingly becoming available in modern data center switches. DCTCP sources extract multi-bit feedback on congestion from the single-bit stream of ECN marks by estimating the fraction of marked packets. In doing so, DCTCP sources react to the extent of congestion, not just the presence of congestion as in TCP[2]. This finer level of control allows DCTCP to operate with very low buffer occupancies while simultaneously achieving high throughput [2].

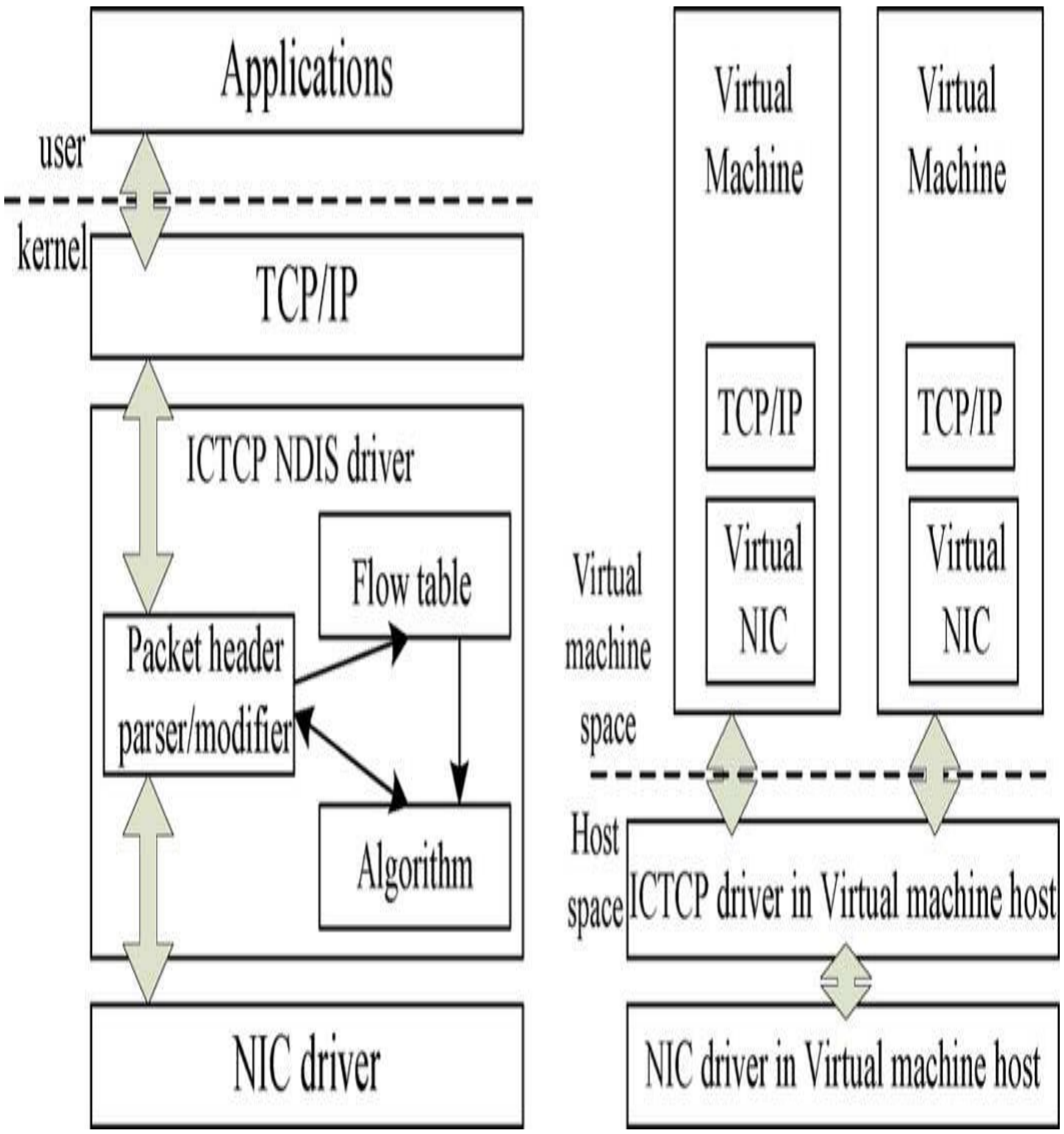
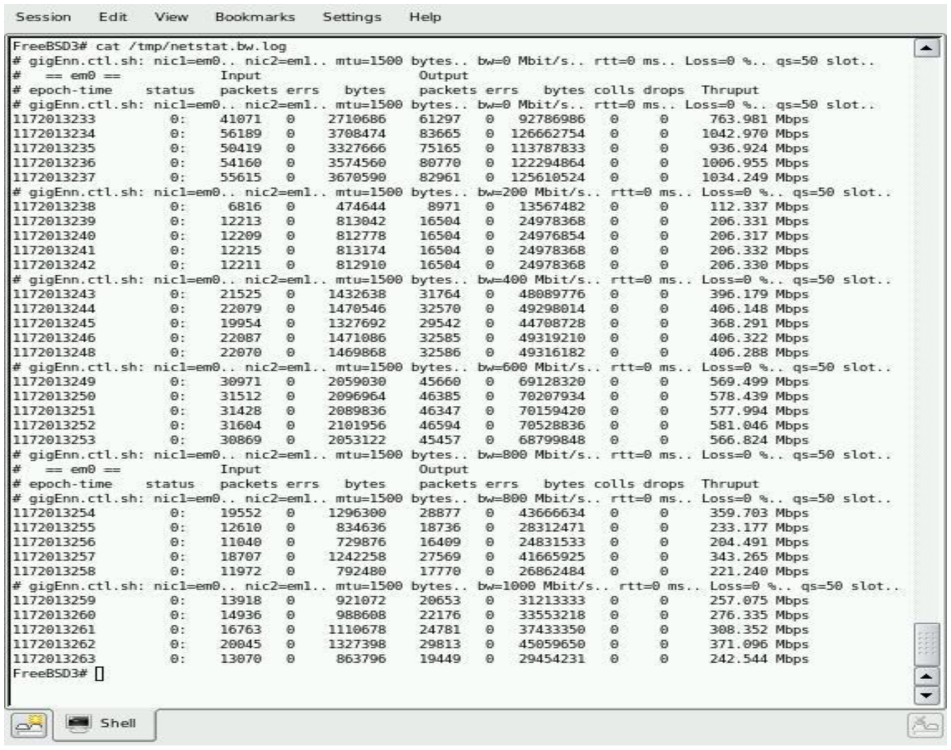


Figure. 3. Modules in ICTCP Driver and Software Stack For Virtual Machine Support.



```
FreeBSD3# cat /tmp/netstat.bw.log
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=0 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
#    == em0 ==            Input                      Output
# epoch-time     status   packets errs    bytes      packets errs    bytes colls drops  Thruput
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=0 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013233         0:     41071   0    2710686    61297   0   92786986   0      0     763.981 Mbps
1172013234         0:     56189   0    3708474    83665   0  126662754   0      0    1042.970 Mbps
1172013235         0:     50419   0    3327666    75165   0  113787833   0      0     936.924 Mbps
1172013236         0:     54160   0    3574560    80770   0  122294864   0      0    1006.955 Mbps
1172013237         0:     55615   0    3670590    82961   0  125610524   0      0    1034.249 Mbps
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=200 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013238         0:      6816   0     474644     8971   0   13567482   0      0     112.337 Mbps
1172013239         0:     12213   0     813042    16504   0   24978368   0      0     206.331 Mbps
1172013240         0:     12209   0     812778    16504   0   24976854   0      0     206.317 Mbps
1172013241         0:     12215   0     813174    16504   0   24978368   0      0     206.332 Mbps
1172013242         0:     12211   0     812910    16504   0   24978368   0      0     206.330 Mbps
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=400 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013243         0:     21525   0    1432638    31764   0   48089776   0      0     396.179 Mbps
1172013244         0:     22079   0    1470546    32570   0   49298014   0      0     406.148 Mbps
1172013245         0:     19954   0    1327692    29542   0   44708728   0      0     368.291 Mbps
1172013246         0:     22087   0    1471086    32585   0   49319210   0      0     406.322 Mbps
1172013248         0:     22070   0    1469868    32586   0   49316182   0      0     406.288 Mbps
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=600 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013249         0:     30971   0    2059030    45660   0   69128320   0      0     569.499 Mbps
1172013250         0:     31512   0    2096964    46385   0   70207934   0      0     578.439 Mbps
1172013251         0:     31428   0    2089836    46347   0   70159420   0      0     577.994 Mbps
1172013252         0:     31604   0    2101956    46594   0   70528836   0      0     581.046 Mbps
1172013253         0:     30869   0    2053122    45457   0   68799848   0      0     566.824 Mbps
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=800 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
#    == em0 ==            Input                      Output
# epoch-time     status   packets errs    bytes      packets errs    bytes colls drops  Thruput
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=800 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013254         0:     19552   0    1296300    28877   0   43666634   0      0     359.703 Mbps
1172013255         0:     12610   0     834636    18736   0   28312471   0      0     233.177 Mbps
1172013256         0:     11040   0     729876    16409   0   24831533   0      0     204.491 Mbps
1172013257         0:     18707   0    1242258    27569   0   41665925   0      0     343.265 Mbps
1172013258         0:     11972   0     792480    17770   0   26862484   0      0     221.240 Mbps
# gigEnn.ctl.sh: nic1=em0.. nic2=em1.. mtu=1500 bytes.. bw=1000 Mbit/s.. rtt=0 ms.. Loss=0 %.. qs=50 slot..
1172013259         0:     13918   0     921072    20653   0   31213333   0      0     257.075 Mbps
1172013260         0:     14936   0     988608    22176   0   33553218   0      0     276.335 Mbps
1172013261         0:     16763   0    1110678    24781   0   37433350   0      0     308.352 Mbps
1172013262         0:     20045   0    1327398    29813   0   45059650   0      0     371.096 Mbps
1172013263         0:     13070   0     863796    19449   0   29454231   0      0     242.544 Mbps
FreeBSD3# []
```

Figure.4. Bandwidth limitations for Round Trip Time using TCP

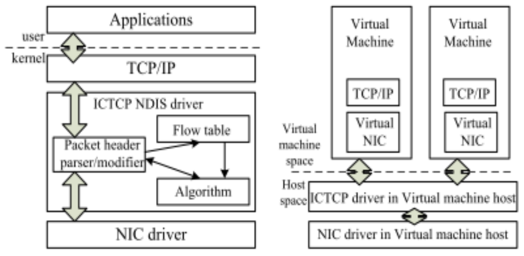


Figure 5: Modules in ICTCP driver and software stack for virtual machine support

Transport Control Protocol (TCP) in cast congestion happens in high-bandwidth and low-latency networks when multiple synchronized servers send data to the same receiver in parallel [2]. For many important data-center applications such as Map Reduce and Search, this many-to-one traffic pattern is common [3]. Hence TCP in cast congestion may severely degrade their performances, e.g., by increasing response time. In this paper, we study TCP in cast in detail by focusing on the relationships between TCP throughput, round-trip time (RTT)[4], and receive window. Unlike previous approaches, which mitigate the impact of TCP incast congestion by using a fine-grained timeout value, our idea is to design an In cast congestion Control for TCP (ICTCP) scheme on the receiver side. In particular, our method adjusts the TCP receive window proactively before packet loss occurs[5].

3.2 Bucket leaky algorithm and rate allocation algorithm

Bucket Leaky Algorithm is an algorithm used to check that data transmissions, in the form of packets, conform to defined limits on band width [4].A fixed capacity bucket, associated with each virtual connection or user, leaks at a fixed rate [3].If this amount of water would cause the bucket to exceed its capacity then the packet does not conform and the water in the bucket is left unchanged [4].

3.3 Bucket leaky pseudo-code

We consider rate allocation algorithm for resolving fundamental problem of bandwidth allocation among flows in a packet-switched network [2].The classical max-min rate allocation has been widely regarded as a fair rate allocation policy. We generalize the theory of the classical max-min rate allocation with the support of both the minimum rate and peak rate constraints for each flow [3].

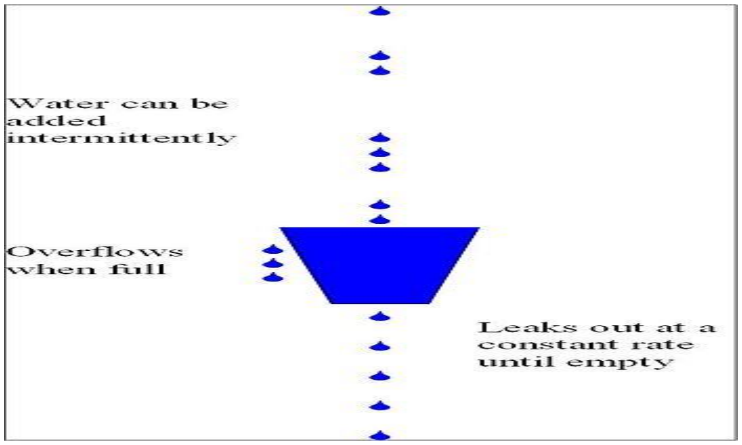

Water can be added intermittently
Overflows when full
Leaks out at a constant rate until empty

Figure 6. Bucket leaky algorithm

The bucket leaky pseudocode is as follows,

```
earches=5
 per_second=60
current_allowed=searches
 last_check = time()
 while process(search_terms):
 # we determine how many seconds since the last search
 time_now=time()
time_passed = time_now - checked_at now
 # and set when our last search was
checked_at=time_now
# add tokens to bucket:
 current_allowed += time_passed * (searches / per_second)
 # check if we have any tokens
 if (current_allowed > searches):
 # we have reached our max. tokens
 current_allowed = searches
 if (current_allowed < 1):
 #partial token, ignore search
```

```
discard_search()
else:
# we have at least one token
do_search()
# and we "spend" the token
current_allowed = current_allowed – 1
```

4. Conclusion

In this paper, we have presented the design, implementation, and evaluation of ICTCP to improve TCP performance for TCP in cast in data-center network s. In contrast to previous approaches that used a fine-tuned timer for faster retransmission, we focus on a receiver-based congestion control algorithm to prevent packet loss. ICTCP adaptively adjusts the TCP receive window based on the ratio of the difference of achieved and expected per-connection throughputs over expected throughput, as well as the last-hop available bandwidth to the receiver. Our experimental results demonstrate that ICTCP is effective in avoiding congestion by achieving almost zero timeouts for TCP incast, and it provides high performance and fairness among competing flows. The number of connections becomes extremely large. Switching the receive window between several value. How to handle congestion while sender and receiver are not under the same switch**.** Use ECN to obtain congestion information.

References

[1].Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in Proc. USENIX FAST, 2008.
[2].V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller,"Safe and effective fine-grained TCP retransmissions for datacenter communication,"
[3].S.Ghemawat,"MapReduce:Simplified data processing on large clusters," in Proc. OSDI, 2004.
[4].C.Guo,G.Lu,D.Li,H.Wu,X.Zhang,Y.Shi,C.Tian,Y.Zhang,and S.Lu,"BCube:A high performance, server-centric network architecture for modular data centers," in Proc. ACM SIGCOMM, 2009.
[5].C.Guo,H.Wu,K.Tan,L.Shi,Y.Zhang, and S.Lu,"DCell: Ascalable and fault tolerant network structure for data centers," in Proc. ACM SIGCOMM, 2008.
[6]. The New Data Center - Brocade FIRST EDITION New technologies are radically reshaping the data center TOM CLARK
[7]. NX-OS and Cisco Nexus Switching: Next-Generation Data Center Architectures, 2nd Edition
[8]. Using TRILL, FabricPath, and VXLAN: Designing Massively Scalable Data Centers (MSDC) with Overlays.

A Brief Author Biography

A. Suresh – Completed Bachelor of Engineering in Computer Science and Engineering and presently doing M.Tech(CSE) at SRM University, Ramapuram.

S. Jayakumar – Working as an Assistant Professor in the Department of Computer Science and Engineering at SRM University, Ramapuram.