DATA CENTER SYSTEMS
RECENT ADVANCES AND ISSUES

CS694: Graduate Seminar Report

Submitted in partial fulfillment of requirements for the degree of
Master of Technology

by
Abhinav Maurya*

under the guidance of
Prof. Bhaskaran Raman†

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai - 400076

Student ID: 10305016, ahmaurya@cse.iitb.ac.in, WWW
†*SynerG, IIT Bombay*, br@cse.iitb.ac.in, WWW

Contents

Abstract

Data center systems are at the heart of massive information systems like the Internet, ensuring availability and scaleability of services that demonstrate great variance in performance metrics. It is therefore necessary and interesting to study these systems in order to ensure that our data-intensive systems can fulfill the goals of efficiency, fault-tolerance, and scaleability.

We begin with the description of contemporary data centers as set forth in [1]. We outline various advances and challenges in recent studies and practical deployments of data centers, in the areas of storage, networking, configuration management, and power management. We also describe the changing face of modern data centers from privately owned infrastructure to virtualized, geographically distributed and publicly leased commodity infrastructure.

Next, we examine the various network topologies that have been proposed for data centers in recent literature. These include CamCube [2], DCell [3], and BCube [4]. We conclude our studies in data center network topologies with a comparison delineated in [5].

We proceed to understand the transport layer issues in data centers. The TCP Incast problem is a particular problem that arises when using vanilla TCP inside data centers. We begin by trying to understand the causes and interactions involved in the occurrence of TCP Incast as explained in [6]. Unfortunately, the problem has only been studied very recently and there is a lack of multiple perspectives on this important issue. We describe the solution to the TCP Incast problem proposed in [7]. We conclude our study of transport layer issues in data centers by understanding DCTCP, a variant of TCP for data centers proposed in [8].

In the final section of this study, we examine recent studies in automatic control of modern massive data centers that are not amenable to manual control. We outline the approach taken by [9] in automating the prediction of performance metrics for MapReduce jobs, and in their design of a realistic workload generator for testing MapReduce optimizations. Another notable work in this area is the online automation of resource allocation in data centers as described in [10].

We conclude the report by listing some advances and issues that could not be covered in the report, and which form an active area of research in data center systems.

Keywords: Data Centers, Information Systems, DCell, BCube, CamCube, TCP Incast, Data Center TCP

1 Introduction

In this report, we explore the advances and issues encountered in the design and deployment of modern data centers. We begin with a description of the various aspects of a contemporary data center as outlined in [1]. Thereafter, we examine various network-layer topologies such as CamCube [2], DCell [3], and BCube [4] proposed in recent literature on data centers; concluding with a cost comparison proposed in [5]. At the transport layer, we seek to understand the pathological TCP Incast problem addressed in [8, 6, 7], peculiar to the use of vanilla TCP in data centers. Finally, we explore the possibility of statistical machine learning based automatic control in data centers [10, 9] by referring to related research carried out at RADLab, UCBerkeley. We conclude with a brief discussion on the active areas of research that could not be covered in the report.

2 Modern Data Center

Data centers lie at the heart of the massive ICT systems deployed by organizations like Google, Yahoo, Amazon, etc. to manage data and to provide online services. The traditional data center could be viewed as a privately owned centralized infrastructure. However, concerns such as better performance and disaster management have led to the emergence of distributed, virtualized data centers (DVDC). A further refinement has been the cloud computing model which utilizes DVDCs to provide pay-as-you-go services to customers. The modern data center is a confluence of various technologies in action - computation, networking, storage, virtualization, configuration control, and power efficiency [1].

2.1 Physical Organization

Main components in a data center are switches, routers, servers, and storage bricks. The physical architecture of a data center is determined by the composition of these components. For example, a data center design may use low-cost L2 switches connected in a mesh network to reduce costs and ensure a high bisection bandwidth.

2.2 Network Infrastructure

Four networking fabrics are prevalent in data centers - Ethernet, InfiniBand, Fiber Channel, and Myrinet. Connection of external clients to data center uses Ethernet, while server-server communication may use Ethernet or InfiniBand. Access between servers and storage has typically used fiber optics, though InfiniBand and Ethernet have also been employed.

2.3 Storage Infrastructure

Three types of storage can be found in data centers. First, Direct Attached Storage (DAS) uses SCSI and SAS, and can provide throughputs of upto 2.4 GBps on a 6 GBps link. Storage Area Network (SAN) and Network Attached Storage (NAS) segregate the storage aspect of the data center from the servers. SAN uses protocols like iSCSI, iFCP, FCIP, and FCoE for communicating with the servers using Fiber Channel. On the other hand, NAS often uses Ethernet and provides higher-level access to memory across the network than SAN. e.g. file or object level access. As a result, it is slower then SAN due to the self-management of storage rather than allowing applications low-level access to data.

2.4 Power Infrastructure

Typically, the external power supply to a data center is 1.33 kV, 3 phase which is stepped down to 280-480 V, 3 phase on premises. This is converted by the UPS from AC to DC for battery storage, and back again from DC to AC for output to the data center. The UPS output of 240V/120V, single phase is distributed by Power Distribution Unit (PDU) to individual servers or chassis, where it is stepped down, converted from AC to DC, and partially regulated. The power is finally delivered to motherboards and converted by VRs into many voltage rails. Due to various losses, overall power efficiency of a data center stands at less than 50%.

2.5 Control Infrastructure

Control and configuration management involves Out-Of-Band (OOB) management for Baseboard Management Controller (BMC), switches, routers, and storage bricks; and InBand (IB) management for server CPU and OS.

3 Data Center Network Topologies

Various data center topologies have been proposed and evaluated in recent literature. Most of these bear resemblance to topologies found in architectures underlying parallel computing. The reasons for rediscovering these topologies in the light of data center systems are not clearly delineated in literature. We shall describe three topologies [2, 4, 3] recently proposed for data center networks.

3.1 DCell

DCell [3] is a Data Center Network (DCN) that has the desirable properties of scaleability, fault-tolerance, and high network capacity. It is a recursively defined DCN architecture where the higher-level DCells contain lower-level DCells. DCell architecture has only one level of mini-switches but a hierarchy of DCells. It differs from other hierarchical architectures like the usual DCN architectures where servers are connected via a tree of core and mini switches, and which have single points-of-failure and bandwidth bottlenecks. Each DCell_k is a collection of DCell_{k-1}s, which can be further considered as virtual nodes consisting of DCell_{k-2}s and so on until DCell_0 which consists of physical servers. The difference is that all DCell_{k-1}s belonging to a DCell_k are fully connected i.e. there is a path from each DCell_{k-1} to every other DCell_{k-1}. This is not the same as a path from each server of each DCell_{k-1} to each server of every other DCell_{k-1}. At the lowest hierarchy, each DCell_0 consists of n servers connected to a single mini-switch. In a DCell_k network, each server has k+1 links, one link connecting it to the mini-switch, and others connecting it to servers in other DCells different from but connected to its own DCell at various levels of the DCell hierarchy.

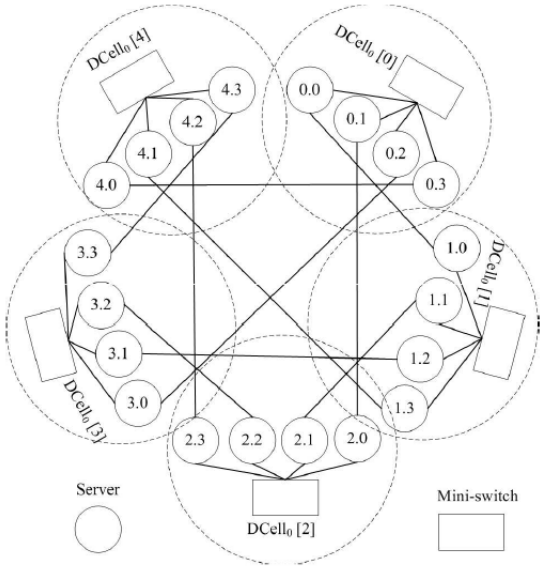

Figure 1: DCell_1 as given in [3]

In a DCell architecture, the number of servers (n) in each DCell_0 is given and is a central parameter to determine the architecture's server density. Each server in DCell_{k-1} contained by DCell_k connects to a unique DCell_{k-1} contained by the same DCell_k. Hence, the number of DCell_{k-1}s is one more than the number of servers in each DCell_{k-1}. The number of DCell_{k-1}s per DCell_k:

$$g_k = t_{k-1} + 1 \tag{1}$$

The number of servers in DCell_k is the product of number of servers in each DCell_{k-1} and the number of DCell_{k-1}s in DCell_k. The number of servers per DCell_k:

$$t_k = g_k * t_{k-1} = t_{k-1} * (t_{k-1} + 1) \tag{2}$$

$$g_0 = 1 \tag{3}$$

$$t = n \tag{4}$$

Each server has an address [a_k, a_{k-1}, ..., a_0]. This can yield a unique ID uid_k for each server in the architecture. In addition, each suffix of length k is also transformable into an address internal to a DCell_{k-1}. Hence, the address can also be written as [a_k, a_{k-1}, ..., a_j, uid_k]. The relationship between address and ID is a bijection. [11] descibes generalized DCells and their properties, including connection rules besides the one used in the algorithm given below.

3.1.1 DCell_k Architecture

1. For each DCell_0, connect all servers to a single mini-switch.
2. For each level l (not equal to 0) of DCell_k network

(a) For each $DCell_j$ at level l

i. For two addresses src = [s_k, s_{k-1}, ... , s_0] = [common_prefix, uid-s] and dst = [d_k, d_{k-1}, ... , d_0] = [common_prefix, uid-d]

A. uids = [as, uid-s]

B. uidd = [ad, uid-d]

C. If as == uid-d and uid-s == ad - 1, connect the two nodes.

3.1.2 Routing

- For two addresses src = [s_k, s_{k-1}, ... , s_0] and dst = [d_k, d_{k-1}, ... , d_0], the routing procedure determines common prefix and hence lowest-level common DCell, say $DCell_j$.
- Since the two nodes' addresses differ in the $DCell_{j-1}$ index i.e. in a_j, the link connecting the two $DCell_{j-1}$s must be found, say (n_1, n_2).
- Hence, the problem now recursively decomposes to finding (src, n_1) and (dst,n_2) paths and combining it with (n_1, n_2).
- Note that these have one more common component in their common prefixes than the previous routing problem. Thus, we are employing a divide-and-conquer approach.

The broadcast mechanism described is basically flooding with duplicate packet detection. Hence, broadcast packet reaches all nodes as long as they are connected to the source.

3.1.3 Fault-tolerant Routing

Local Reroute: If two nodes are in different $DCell_{j-1}$ and same higher-level DCells and the link between the two DCells fails, source $DCell_{j-1}$ reroutes to a proxy $DCell_{j-1}$ which can then route to destination $DCell_{j-1}$.

Local Link-State Routing: Local link state routing is used for intra-$DCell_b$ communication and local reroute for inter-$DCell_b$ traffic.

Jum-Up for Rack Failure: If an entire $DCell_b$ fails, then local reroute may not work. In such a case, the packet must be routed to other DCell higher up in the hierarchy, which increases path length.

3.1.4 Properties

- $((n+0.5)^{2^k}-0.5) < t_k < ((n+1)^{2^k}-1)$ Thus, the number of servers in $DCell_k$ network is bounded both ways by k doubly exponentially. The number of links per server is k+1. The link density and hence availability is very high. The number of servers and hence the links increases rapidly with k. Hence, DCell is scaleable.
- The bisection width of a DCell is larger than $\frac{t_k}{4*logt_k}$. Hence, the network is highly fault-tolerant.
- The high number of links provide scaleability and fault-tolerance, and high network capacity - the goals of the architecture. However, this also means that the costs and maintenance overheads of immense wiring can be prohibitive.

3.1.5 Results

Simulations were carried out with $DCell_3$ topology and n={4,5}. It is observed that DCell performs very close to shortest path routing on the metric of path failure ratio versus node failure ratio. The testbed on which experiments have been carried out is a $DCell_1$ with n=4 and has topology as given in figure (1). In the aspect of tolerance to node and link failures, the TCP throughput reverts to the best value in a few seconds after the recovery from the failure. In the throughput experiment, each server sent 5 GB to each of the other servers over a TCP connection. The maximum aggregate throughput noted in DCell was 9.6 Gbps, as against 3.6 Gbps in a two-level tree topology. The duration of the transfer was also better for DCell - 2270 seconds against 4460 seconds taken in the tree topology.

3.2 BCube

BCube [4] is a fault-tolerant and load-balancing DCN and accelerates representative bandwidth-intensive applications. BCube uses source routing and bears similarity to a generalized hypercube topology.

3.2.1 $BCube_k$ Architecture

The BCube architecture consists of servers and switches connected in the following manner:

- Number of servers = n_{k+1} with (k+1) ports each.
- Number of switches = n_k (with n ports each) at each of the (k+1) levels.
- Server port address: $a_k...a_1a_0.[0\text{-}k]$
- Switch port address: $b_k...b_1b_0.[0\text{-}k]$
- Connect $s_{k-1}...s_1is_{l-1}...s_0.l$ server port to $s_{k-1}...s_1s_{l-1}...s_0i.l$ switch port.

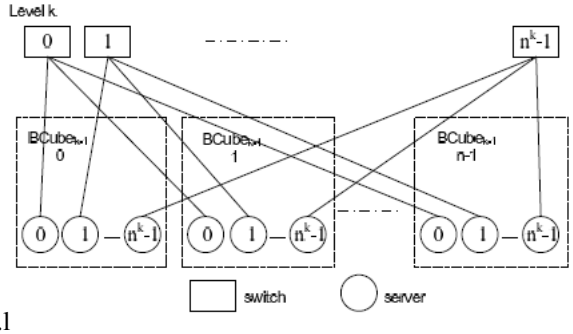

Figure 2: $BCube_k$ as given in [4]

3.2.2 Routing

- Assume $a_k...a_l...a_1a_0$ and $a_k...b_l...a_1a_0$ differ in only one digit.
- $a_k...a_l...a_1a_0.l$ server port connects to $a_k...a_1a_0a_l.l$ switch port.
- Switch changes port from $a_k...a_1a_0a_l.l$ to $a_k...a_1a_0b_l.l$
- Server port $a_k...b_l...a_0a_l.l$ receives packet from $a_k...a_1a_0b_l.l$
- Multiple bit changes are handled by correcting one bit at a time.

3.2.3 BSR (BCube Source Routing)

Routing is done by BSR at source and the path embedded in the BCube header sandwiched between TCP/IP and MAC. Source probes the network for the (k+1) parallel paths from itself to destination. The intermediate servers along the parallel paths write out a metric (e.g. bandwidth) into the probe packet. If a link or a server on the route is down, the on-path server detecting the failure sends a failure probe response back to source. The destination creates a response probe confirming existence of parallel path and sends it back to source. The entire list of servers to be traversed in the BCube is encoded using NHA (Next Hop Array) in the BCube header of each packet sent from one server to another inside the BCube. This reduces the complexity of the forwarding system which must perform only a header and a table lookup to send an incoming packet on its way.

The source maintains a database of discovered failed links, which is flushed based on timeouts. The source maintains the list of parallel paths from source to destination for each flow. If the path currently being used fails, the source switches to a (possibly suboptimal) path from the list of available edge-disjoint paths. This avoidance of failure-based probing for optimal paths reduces the loss of packets due to original path failure. If a more optimal path is detected at the next periodic path probing exercise, the flow switches to the more optimal path.

3.2.4 Speedups

- One-to-one: Throughput can be optimized using k+1 edge-disjoint parallel paths.
- One-to-several: Optimizations are effected using edge-disjoint complete graphs of communicating servers.
- One-to-all: Speedup is achieved by constructing (k+1) edge disjoint spanning trees for each source.
- All-to-all: Aggregate Bottleneck Throughput (ABT) for $BCube_k$ is n(N-1)/(n-1) which is better than N of fat trees.

3.2.5 Properties

- Diameter of $BCube_k$ is (k+1).
- Parallel paths between 2 servers = k+1: h(A,B) from bits that differ and k+1-h(A,B) from bits that have not changed.
- Maximum path length in a $BCube_k$ is (k+2).

3.2.6 Results

Simulation performance is comparable to fat-tree for server failure, and is much better than fat-tree for switch failure, since BCube provides many parallel paths between two nodes compared to fat-tree.

In real testbed experiments, BCube is compared against a two-level tree structure of switches. In each of the following cases, each server sends 10 GB. As the MTU increases, the CPU overhead for packet forwarding decreases. For one-to-one traffic, 1.93 Gbps was the achieved throughput compared to 0.99 Gbps in the tree structure For one-to-several traffic, BCube uses 47.9 sec to finish the transfer instead of 89 seconds in a tree structure due to edge-disjoint complete graphs. For one-to-all traffic, BCube's throughput was 1.6 Gbps as against 880 Mbps in a pipelined single-tree structure. Finally, for all-to-all traffic, figures stood at 750 Mbps for BCube as against 260 Mbps for tree structure.

3.3 CamCube

CamCube [2] is a data center system that provides applications the choice to direct routing as per their needs, in addition to a default routing mechanism. CamCube's main focus is on studying the changes to DCN performance when the routing in direct-connect DCN is done, taking into consideration application specific concerns. Authors observed that due to flexibility of the routing protocol in CamCube API, developers made application specific routing choices, which helped the applications gain efficiency and minimize the traffic due to awareness of application requirements. Four examples of such application specific routing choices are described in the paper.

CamCube uses the 3D torus topology widely used in parallel computing architectures. The 3D coordinate space in which this topology rests and the functionality of the CamCube API to send packets to neighbors one hop away comprises the services provided by CamCube to application. Symbiotic routing refers to the fact that the various application specific routing protocols run simultaneously and benefit from the base routing protocols ability to handle exceptions in routing like voids in the coordinate space. Each service running in the data center has a ServiceID. This is present as a two-byte header on each packet to tell a server which application can process the packet.

3.3.1 Service Queueing

CamCube polls each application for a packet to send and lets the services manage their internal packet queues themselves. This can be easily extended to weighted sharing of bandwidth by changing polling frequency.

3.3.2 Core Services by CamCube API

- Services provided by CamCube API bear semblance to KBR API.
- Implicit knowledge by servers of the homogenous direct-connect 3D torus topology.
- Exposing a server's coordinates and that of its one-hop neighbours.
- Monitor liveness of the links to one's neighbours and liveness of servers in the system.
- A simple multi-hop routing protocol based on link state routing.
- Routing service that routes around failure voids using base link-state routing.
- Routing service that drops packets destined to failed or unreachable servers.
- Routing service that can perform key-based routing and server-based routing.
- Routing service that ensures key-coordinate consistency in case of server failures.

3.3.3 Experiments

The results are reported for four services simulated for an 8000 node system and actual runs on a 27 node testbed.

TCP/IP service: Use of link-disjoint routing protocol improves performance of TCP/IP tunnelled over the CamCube routing protocol.

Algorithm: For any two nodes in the wrapped 3D torus topology, communication throughput can be improved by using link-disjoint paths. Three such disjoint paths can be obtained by simply traversing the 3D path using the following axis orders: x-y-z, y-z-x, z-x-y.

Factor	BCube	DCell	CamCube
Network Architecture	2 layer hierarchy of switches	Single layer of switches	No switches
Protocol Stack	TCP/IP	TCP/IP	CamCube API
Ability to use Multi-path Optimizations	Y	Y	Y
Ability to do On-path Modifications	Y	Y	Y
Service Involvement in Optimizations	N	N	Y

Table 1: Comparison of [2, 4, 3]

VM Distribution: The distribution of large files using a multicast tree can be made efficient by having path segments of the multicast tree shared whenever feasible. This is done by restricting the edges that join requests to the multicast tree can traverse.

Algorithm: The server holding the VM distribution is considered the center of the 3D cube. The cube is hierarchically and recursively split into 3D mini-cubes, the single level-0 cube degenerating into 8 level-1 mini-cubes which are further split into 64 level-2 mini-cubes, and so on until the mini-cubes are indivisible. Each join request from a server/node to the multicast tree traverses only the edges of the mini-cubes and ascends the hierarchy at the mini-cube corners. This leads to fewer edges in the multicast tree and improves multicast efficiency.

Cache Service: Servicing of cache requests. The query is input in the form of a key.

Algorithm: Cache requests are forwarded to replica key coordinates. In case the replicas do not have a copy, the request is forwarded to primary key coordinate. The response is routed back via the replica key coordinates which are populated with the response. This is better than servicing by on-path servers which may or may not have secondary replicas of the object.

Aggregation Service: The assignment operation of the MapReduce can make better use of all links if the servers select the axes order to send on based on the destination coordinates.

Algorithm: At the end of the map phase of MapReduce, each server has key-value pairs; a single key may occur in key-value pairs across multiple servers. Now, each key is assigned to a server. Servers act as reducers i.e. they are supposed to collect all the key-value pairs for a key assigned to them and perform an aggregation operation to generate a single key-value pair. This aggregation can be performed on-path. For each key, a hash randomly determines the axes order that transfer of key-value pair from any source to the designated key coordinate must follow. Since all transfers for a key follow the same deterministic axis order (say x-y-z), the data aggregates first on the y-z plane and then along the z axis. The servers along the plane and the axis can perform on-path reduce operations and reduce the data and aggregation load transferred to the destination key coordinate. Since different keys follow different axis order, links along all axes are used, thereby not constraining the solution.

TCP/IP and VM Distribution Services have opposite objectives, while the principle used in Aggregation Service is similar to that of VM Distribution Service. The extended routing service is used by services to specify their own routing protocol in the form of a function F from a key to a destination key coordinate.

3.3.4 Results

- Aggregate link stress is lower in all simulations for the application-specific routing compared to base routing.
- In case of individual link stress, the 99 percentile and maximum stress links reported equal or lower stress for the application-specific routing compared to base routing. The only exception was the aggregation service in which on-path aggregation is not possible.
- Experiments suggest that CPU utilization increases from 22.01% to 25.35% when the application specific routing protocols are used.

3.4 Comparison

[5] proposes a methodology for predicting and comparing costs of a data center infrastructure, and uses the methodology to compare four representative data center network architectures.

The paper identifies four representative data center network architectures:

1. **SW-FatTree:** This is a switch only architecture. e.g. simple tree topology, Clos topologies like fat-tree.
2. **HY-BCube:** This involves both switches and servers in routing. However, the routing costs are borne more by switches. e.g. BCube and DCell.
3. **HY-deBruijn:** This involves both switches and servers in routing. However, the routing costs are borne more by servers. e.g. modified SRV-deBruijn in which intra-rack de Bruijn graphs are replaced by a single ToR (Top of Rack) switch.
4. **SRV-deBruijn:** Here routing is done solely by servers. e.g. The only example so far is CamCube which was published in 2010. The authors argue that CamCube is very inefficient in latency and generalize the 3D torus topology of CamCube to n dimensional k-base de bruijn graph, in which two nodes are based on similarity of their address.

3.4.1 Proposed Methodology

1. **Equalize latency (path lengths):** By changing various parameters like number of levels, ports per switch, the fanout at every server; the average path lengths of various topologies are made equivalent. The resultant topology generated for simulation must be feasible in real life.
2. **Equalize capacity:** For a target input traffic, high-level simulations generate the load at each switch and server. The network resources are provisioned such that this load is satisfied by the network.
3. **Compare costs:** The costs comprise the switch port costs (if switches are present in the architecture), server NIC port costs, CPU core costs (if servers participate in the routing), cabling costs, and power costs. A major consideration is if some cores are dedicated to network routing at a server (reserved model) or if the core time is shared by computational and routing tasks (shared model).

3.4.2 Results

The results are from simulation for a topology of upto 60000 servers. If reserved model is considered, HY-BCube is the least costly of the four, followed by SW-FatTree and HY-deBruijn. If the shared model is considered with average network utilizations of 10% and 20% for routing tasks, the HY-deBruijn is the least costly. The authors conclude that the hybrid designs are cheapest both in reserved and shared models. Amongst these designs, the savings depend on the extent to which cores can participate in network routing tasks. In a shared model, the server only architectures are cheaper than switch only architectures, since cores are not reserved for network tasks alone and their costs is only partially counted towards network infrastructure costs. In a reserved model, the switch only architectures are cheaper than server only architectures, since cores are reserved for network tasks. Also, the ports required per server increases thereby increasing the costs of NIC per server. The cabling alone is just 3-8% of the equipment cost. Cabling and power consumption costs together are around 10-16% of the total costs and hence not major considerations for a choice between architectures. For each topology, asymptotic bounds have been suggested for the prices of switches. NICs and cores.

3.4.3 Predicted Future Trends

µ-Switches: These are located within server's PCIe or I/O hub. They do the task of filtering packets that are of no use without requiring CPU intervention.

Hybrid Cores: In this case, two types of cores can be present in the server. Sophisticated powerful cores can do the computation and simpler cores can do the task of routing.

4 TCP Issues in Data Centers

TCP Incast problem is defined as the degradation of goodput (application throughput) when multiple senders communicate with a single receiver in a high throughput, low latency network with shallow switch buffers often found in commodity switches.

4.1 Fine-grained TCP for Data Centers

To solve this problem, the authors of [7] suggest reducing or eliminating the RTO_{min} parameter, using high-resolution timers to maintain RTT and RTO timer values at microsecond resolutions, randomizing the timeout to desynchronize the flows, and disabling the delayed ACK feature.

4.1.1 Analysis

In barrier synchronized requests, a client makes the request to multiple servers for data portions. The client cannot proceed with another request unless all previous parallel requests have been answered. In vanilla TCP designed for WAN architecture, the RTO timer is calculated as a function of the RTT samples. It is lower-bounded by RTO_{min}, which is usually 200 ms. This minimum value helps avoid spurious timeouts when the RTT suddenly increases.

In a DCN, the latencies are of the order of microseconds. As a result, in the TCP Incast scenario, the flows whose packets are lost must wait an inordinately high amount of time compared to the network latency to retransmit their packets. In a barrier synchronized request scenario, this can lead to synchronized retransmissions and losses, leading to a loss in goodput by an order of magnitude. The above problem can be avoided by setting the RTO_{min} to 200 µs. This ensures that the effective RTO is not too high to cause long link idle times. However, in future DCN, where the latencies will be further reduced due to 10 Gbps links and the RTT will be around 20 µs, the 200 µs RTO_{min} will again lead to TCP Incast. Hence, the conclusion that the RTO_{min} should be of the order of the network latency. This is required to ensure that RTO_{min} does not delay detection of TCP Incast losses and subsequent retransmission.

In order to maintain timers with microresolution granularity, the authors describe kernel changes that allow them to utilize the GTOD framework that uses CPU cycle counter and provides nanosecond resolution. This is used for timestamping and calculating RTT. Also, hrtimer interface using HPET (High Precision Event Timer) hardware is used for maintaining TCP timers such as RTO timer. Reducing RTO_{min} does not prevent the retramsmissions from coinciding leading to a throughput drop, since the RTO values are chosen deterministically. The authors suggest adding a random factor to the timeout value calculation as given below, which explicity desynchronizes the retramsmissions of flows.

$$timeout = (RTO + rand(0.5) * RTO) * 2^{backoff} \tag{5}$$

4.1.2 Results

- Simulations were carried out by varying RTO_{min} and the number of senders, with 1 Gbps links and ~100 µsec RTT. High RTO_{min} leads to goodput loss. Reduction in RTO_{min} leads to increase in goodput upto a certain point at which the RTO_{min} is of the order of the network latency i.e. 1 ms. Below this, the goodput does not increase significantly.
- Real experiments were carried out on 2 clusters of 16 and 48 servers connected via a switch, with 1 Gbps links and ~100 µsec RTT. The Linux kernel is 2.6.28 and the experiment is conducted for 4, 8, 16 senders.
- 200 µsec RTO_{min} shows the highest goodput for lower number of servers, and is overtaken by the case without any RTO_{min} when number of senders exceeds ~100.
- Experiments were also carried out with the modified TCP to see the effect of reduction in RTO_{min} on wide-area flows using torrent uploads. Surprisingly, the throughputs of the unmodified and modified TCP servers are almost the same, indicating that the reduction in RTO_{min} does not lead to considerable spurious retransmissions over WAN.

4.2 Understanding TCP Incast

The work in [6] reproduces results of earlier work described in [7] to demonstrate generality of TCP Incast problem. It provides an incomplete quantitative model to explain some parts of the empirical curve, and qualitative explanations for the remaining curve characteristics that are not covered by the quantitative model. It also suggests minor, intuitive modifications like reducing RTO_{min} already suggested by [7]. Authors tried out setting a small multiplier to RTO exponential backoff, setting a randomized multiplier for RTO exponential backoff, randomizing the minimum TCP RTO timer value, and decreasing the minimum TCP RTO timer value. Only the last approach worked.

4.2.1 Observations

Smaller minumum RTO values lead to larger initial goodput minimum. The initial goodput minimum occurs at the same number of senders for all values of minimum RTO. Larger RTO minimum values lead to the goodput local maximum occuring at higher number of senders. After the goodput local maximum, the rate of goodput decrease with the number of senders is independent of the minimum RTO values. All observations have been made for a fixed fragment size, variable block size workload.

4.2.2 Disabling Delayed ACKs

It leads to a higher and less stable congestion window due to immediate ACKs overdriving the congestion window. This causes an increased number of smoothed RTT spikes inspite of the mean smoothed RTT remaining the same, resulting in frequent, unnecessary congestion timeout events. Hence, the authors of [6] conclude that delayed ACKs should not be disabled. This observation is different from the one in [7], where it is suggested that delayed ACKs can cause a throughput drop in data center environment and should be disabled.

4.2.3 Quantitative Model

The authors of [6] define D as the total amount of data, L as the total transfer time without RTO events, R as the number of RTO events, r as the value of RTOmin, S as the number of senders, and I as the inter-packet wait time. The throughput for a single sender (G_1) and for all senders (G_s) is given as

$$G_1 = \frac{D}{(L + R * r)} \tag{6}$$

$$G_s = \frac{S * D}{(L + R * r)} \tag{7}$$

R is estimated using a piecewise linear approximation of empirical data as follows:

$$R = \begin{cases} \frac{35}{10} * S & \text{if S<=10} \\ 35 & \text{if S>10} \end{cases} \tag{8}$$

L is the sum of packets' transmission time and the inter-packet wait time. Hence, it is given as:

$$L = \frac{D}{Bandwidth} + \frac{D}{averageMSS} * I \tag{9}$$

I is estimated using a piecewise linear approximation of empirical data as follows:

$$I = \begin{cases} \frac{4.5}{10} * S & \text{if S<=10} \\ 4.5 & \text{if S>10} \end{cases} \tag{10}$$

4.2.4 Qualitative Refinements

Initial Goodput Decrease: As S increases, R increases. This causes more RTO events and spurious retransmissions leading to goodput loss.

Goodput Recovery: As the number of senders increases, the RTT and RTO estimate of each sender also increases, since all senders share the same queue. This leads to lesser RTO events and hence goodput increases.

Further Goodput Decrease: As senders increase, RTO keeps increasing. After a certain RTO increase, there would be interference between the senders transmitting after an RTO timeout and senders that are transmitting because they are not in an RTO state.

4.3 Data Center TCP (DCTCP)

Transport protocol design can make certain requirement assumptions in the data center environment that cannot not be made generally in the Internet due to end-to-end principle. These assumptions such as low latencies, high throughput, bursty traffic, and low packet losses allow for customizing the TCP for much better performance in the homogenous environment of data centers.

DCTCP [8] is a TCP variant for the low latency, high throughput, bursty data center environment that notifies the sender of congestion using ECN and reduces the congestion window size in proportion to the extent of congestion as indicated by the receipt of ECN packets.

According to authors, the following typical traffic classes are reported in data centers: query traffic (very short, latency critical flows; between 1.6 kB to 2 kB), short message flows (time-sensitive; 50 kB to 1 MB), and large data flows (throughput-sensitive; 1 MB to 50 MB). Thus, application-level query and message traffic is latency sensitive while data traffic is utilization sensitive.

4.3.1 Problems with TCP

The following problems are reported with TCP in data center environment:

Incast: In the Partition/Aggregate pattern, a receiver requests for data simultaneously from multiple receivers. As a result, the buffer queue length at the switch port of the receiver increases, thereby causing timeouts. TCP interprets this worngly as network congestion and reduces cwnd thereby reducing network utilization drastically.

Queue Buildup: When a queue at a port is shared by short as well as long flows, the shorter flows experience latencies. Since performance over short flows (application-level query traffic and messages) is based on latency, the performance degrades. This is because of delays in the buffers being shared by the short and long flows.

Buffer Pressure: Switches are often shared memory switches i.e. all ports share the same memory. This reduces costs under the assumption of statistical multiplexing. However, long flows in a DC are often stable and do not utilize statistical multiplexing. As a result, a short flow at one port may be coupled with a long flow at another port of the same switch due to shared memory queue buffers.

4.3.2 DCTCP Algorithm

1. Simple marking of CE codepoint at switch: Based on a threshold K of the *instantaneous* queue, each switch can mark a packet with a CE codepoint. This indicates to the receiver that congestion has occured.
2. ECN-Echo at the receiver: The receiver uses a combination of immediate and delayed ACKs to notify the sender about congestion. The delayed ACKs are used to convey ECN status of a certain number of packets at a time, and reduce pressure on the sender. The immediate ACKs are used when the ECN status of packet most recently received at the sender changes compared to its previous packet.
3. Controller at the sender: The sender maintains a parameter α indicating the level of congestion, which is updated every RTT. The sender receives the ACKs from the receiver and changes cwnd according to congestion. Given that F is the fraction of packets marked for congestion during the last data window, the formulae used by the controller for computing α and cwnd are as follows:

$$\alpha := (1-g) * \alpha + g * F \tag{11}$$

$$cwnd := cwnd * (1 - \alpha/2) \tag{12}$$

The notification mechanism of DCTCP is similar to AQM/RED, except that DCTCP detects congestion using instantaneous queue length and reports it using delayed ACK. Also, the backoff is based on the extent of congestion conveyed by ECN-Echo.

4.3.3 Results

The experimental testbed consists of 45 servers connected via Triumph switch with 1 Gbps links. All three kinds of traffic were generated: query, short message and background traffic. The RTO_{min} was set to 10ms for both TCP and DCTCP. In TCP, 1.15% queries suffered from timeouts, whereas in DCTCP, no queries suffered timeouts. Short-message traffic sees a 3ms/message latency reduction at the mean of the completion delay distribution and 9ms/message at the 95^{th} percentile. DCTCP consistently offered a better completion delay than TCP for background traffic requiring high throughput.

5 Automatic Control in Data Center Environment

5.1 Automated Resource Provisioning

Automatic provisioning is not popular in data centers inspite of the complexity of the system due to two reasons. First, simple linear models and queueing models are not realistic for complex systems. Second, previously employed models were not robust to changes often observed in DC environments. [10] describes a system for automatic provisioning of resources in data centers.

There are four components to the proposed automatic control system: resource control loop, statistical performance model, change-point detection, and control policy simulator.

5.1.1 Resource Control Loop

It is executed every 20 seconds and does the following things:

1. Predict workload for the next 5 minutes based on recent 15 minutes by linear regression/other methods.
2. Input {workload, required performance} to the performance model to obtain required servers.
3. Add or subtract servers using hysteresis co-efficients to prevent oscillations i.e. do resource scheduling.

5.1.2 Statistical Performance Models

The performance model is trained using data of the form {workload, #servers, observed performance}. The model predicts performance mean using smoothing spline non-linear regression and variance using LOESS regression using {workload, #servers} as input. The model can predict #servers using {workload, required performance}.

5.1.3 Change Point Detection

If the performance predicted by model and performance observed in practice deviate, then we can say that the performance model no longer correctly reflects the current scenario.The shift in mean or increase in variance of residual distribution (difference between the measured and predicted performance) indicates change point. We can use statistical hypothesis testing (p-test) to determine if change point should be recognized. For our case, the p-test would be formulated as follows: Given a performance model, what is the probability of predicting performance at least as extreme from the predicted mean as the observed performance?

5.1.4 Control Policy Simulation

Simulations for optimal control policy was done using real past workloads. The following procedure was adopted:

1. Use performance model to get #servers from input {workload, required performance}.
2. Use particular α and β to play out the gain scheduling and get #servers scheduled.
3. Use performance model to predict performance with input {workload, #servers}
4. Use hill climbing to optimize α and β.

5.1.5 Results

The experiment used Cloudstone Web 2.0 benchmark written in Ruby on Rails and executed on Amazon's EC2 platform. The authors used control policy simulator to find optimal α=0.9 and β=0.01. Using the above hysteresis parameters, the authors ran the benchmark workload with no SLA violations and few controller actions to modify the servers. The authors found that at the begin and end of a three hour long performance anomaly, the p-test value approached zero. This conforms that p-test can be used to detect change-points for training new performance model. However, no online training has been reported in the paper to deal with the information conveyed by a change point detection using p-tests.

5.2 Prediction of MapReduce Performance Metrics

Research described in [9] provides a prediction model using KCCA (Kernel Canonical Correlation Analysis) to predict execution time of MapReduce jobs, and a workload generator to synthesize realistic workloads to evaluate MapReduce optimizations. It explores applying Kernel Canonical Correlation Analysis (KCCA) to predicting performance metrics of MapReduce jobs given a feature vector of job characteristics. KCCA can and should be applied to MapReduce because MapReduce has a homogenous execution model required to apply KCCA. Also, KCCA predicts multiple performance metrics simultaneously and captures interdependencies between the metrics by finding correlation between multidimensional feature vectors.

5.2.1 Training

KCCA modeling for correlation is done between pre-execution feature vector of job characteristics (x_k) and post-execution feature vector of performance metrics (y_k). KCCA Modeling is done by converting x_k to K_x and y_k to K_y using kernelization. Then, K_x and K_y are mapped to smaller subspaces of the same dimension by multiplying with transforming matrices of basis vectors A and B, such that the correlation between the mapped data points in K_xA and K_yB is maximized. The above formulation of KCCA reduces to a generalized eigenvector problem shown in figure (3).

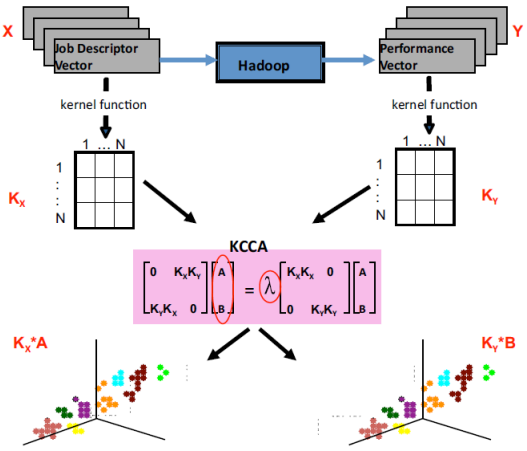

Figure 3: KCCA applied to K_x and K_y as given in [9]

The feature vector of job characteristics consists of job configuration i.e. number and location of map and reduce tasks in the MapReduce job, map input bytes, Hadoop Distributed File System (HDFS) read bytes, and locally read bytes. The feature vector of performance metrics consists of map time, reduce time, overall execution time, map output bytes, HDFS written bytes, and locally written bytes.

5.2.2 Prediction

Once KCCA has found the transformation matrices A and B, it can do metric prediction To predict metrics for a MapReduce job with job feature vector x, following procedure is followed:

1. Map x to xA.
2. Find three nearest neighbors of xA in the reduced subspace.
3. Find the corresponding performance metrics of these neighbors.
4. Take the average of these performance metric vectors to predict performance metrics corresponding to the job characteristic feature vector x.

5.2.3 Workload Generation Framework

To capture statistical summary of production MapReduce trace, we need to capture the distributions of job count by job names and inter-job arrival times. Also, for each job, we need to capture the distribution of job input sizes scaled by thee number of nodes, the input/shuffle ratio, and the shuffle/output ratio. These distributions captured from production MapReduce traces collected from master daemons can be used to generate realistic workloads for testing MapReduce optimizations.

5.2.4 Results

Authors report accuracies of 0.87, 0.84, 0.71, and 0.86 for the metrics of execution time, map time, reduce time, and the number of bytes written by the job. These jobs are generated by HiveQL, an SQL-like data warehouse interface built to run on top of Hadoop. Authors also report accuracies of 0.93, 0.93, and 0.85 for the metrics of execution time, map time, and reduce time for MapReduce jobs generated by ETL processes running on top of Hadoop. A suggested modification for future work is to augment the job feature vector with map/reduce function identifiers, language of the map/reduce functions, and resource consumption levels. Other suggestions are to use different Hadoop workloads, job types, and resource types to test the generality of the method.

6 Conclusion

The seminar readings have concentrated mainly on the network layer [2, 3, 4, 5] and transport layer [6, 7, 8] issues in the data center environment. Though some of the papers read have been about automatic control in data centers [9, 10], the treatment of these papers has been generally towards the solution of automatic control problems in complex systems, data center systems being an apt example for the application of these solutions.

Some notable works in the area of data center research could not be included due to lack of time and space. Particularly, the MapReduce framework described in [12] has had a defining impact on the programming model used in data centers, and can be arguably regarded to be the C/C++ of the data center world.

In the solution space of TCP Incast, [13] provides some fresh insights by trying to adopt a preventive approach to solving the TCP Incast problem. The authors suggest controlling the receiver window to prevent the occurrence of TCP Incast instead of using fine-resolution RTO timers as propounded in some earlier works described in this report. We would also like to point out an analysis of DCTCP described in [14].

The literature available on data centers is vast and growing by the day. The author of this report hopes that he has covered significant work that will have a lasting impact on future research carried out in the area of data centers.

References

[1] K. Kant, "Data center evolution: A tutorial on state of the art, issues, and challenges," *Computer Networks (Elsevier)*, vol. 53, pp. 2939–2965, 2009.

[2] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," SIGCOMM, August 30–September 3 2010.

[3] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: A scalable and fault-tolerant network structure for data centers," SIGCOMM, August 17–22 2008.

[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," SIGCOMM, August 17–21 2009.

[5] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica, "A cost comparison of data center network architectures," ACM CoNEXT, 2010.

[6] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in data center networks," WREN, August 21 2009.

[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 data center communication," SIGCOMM, August 17–21 2009.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," SIGCOMM, August 30–September 3 2010.

[9] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," ICDE Workshops, 2010.

[10] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," HotCloud: USENIX Workshop on Hot Topics in Cloud Computing, 2009.

[11] M. Kliegl, J. Lee, J. Li, X. Zhang, D. Rincon, and C. Guo, "The generalized dcell network structures and their graph properties," tech. rep., Microsoft Corporation, 2009.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[13] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp," in *ACM CONEXT*, Association for Computing Machinery, Inc., 2010.

[14] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of dctcp: Stability, convergence, and fairness," in *SIGMETRICS*, June 7–11 2011.