EFFICIENT CROSS-LAYER ROUTING AND CONGESTION CONTROL ARCHITECTURES FOR DISTRIBUTED SYSTEMS

BY

DEBESSAY FESEHAYE KASSA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Klara Nahrstedt, Chair
Professor Brighten Godfrey
Professor Steven S. Lumetta
Professor Sonia Fahmy, Purdue University

Abstract

Major distributed systems such as the *Internet*, *datacenter* and hybrid P2P networks share a common known challenge of finding an *optimal path* to transfer content from a source to a destination and the *optimal rate* at which content is transmitted. In general networks such as the Internet, per user, there is usually one possible content source/destination such as a web server. There can be multiple possible paths to/from the destination/source (server).

In datacenter networks which usually have a tree structure and in hybrid Peer-to-Peer (P2P) networks, there can be multiple possible servers at which content can be stored and from which content can be retrieved. Multiple possible servers (sources/destinations) translates into multiple possible paths to/from a content destination/source. Finding an optimal path to/from a destination/source requires efficient *congestion control* and *routing* schemes.

The transmission control protocol (TCP) is the major congestion control protocol in the Internet. TCP and its variants have the drawback of not accurately knowing rate share of flows at bottleneck links. Some protocols proposed to address these drawbacks are not fair to short flows, which are the majority of the Internet traffic. Other protocols result in high queue length and packet drops which translate into a high average flow completion time (AFCT).

The currently major deployed intra-domain routing algorithm is the Open Shortest Path First (OSPF) [1]. OSPF uses a simple heuristic routing metric (link weight). The routing metric used doesn't properly take into account the latest status of the network. Other traffic engineering schemes such as the TeXCP proposed to address the routing issues of existing schemes also fail to find a globally (domain level) optimal route. Besides, they incur additional probing and path reservation packet overheads.

Recently deployed datacenter network architectures rely on random server and hence path selection in the attempt to ease congestion. However such random selection of paths can result in serious congestion and content transfer delay. This can for instance be caused by large content transfers (elephant flows) which take a long time to finish. In this case a random path selection can add to the congestion caused by elephant flows. Existing cloud datacenter architectures such as the Google File System (GFS) and the Hadoop File

System (HDFS) rely on a single name node server (NNS) to manage metadata information of which content is stored in which server. A single NNS can be a potential bottleneck and a single point of failure.

Hybrid P2P content sharing can result in significant scalability gains in addition to assisting content distribution networks (CDNs). However, currently proposed CDN and P2P hybrid schemes do not provide accurate, fair and efficient incentives to attract and maintain more peers. Besides, they do not use efficient prioritized congestion control and content source selection mechanisms to reduce content transfer times.

In this thesis, we present the design and analysis of *cross-layer congestion control and routing* protocols to address the above challenges of major distributed systems. Our schemes derive an efficient rate metric which sources use to set their sending rates and which routers/switches use as a link weight to compute an optimal path. Among other contributions our rate and path computation schemes achieve network level max/min fairness where available resources are quickly utilized as long as there is demand for them. Our schemes have prioritized rate allocation mechanisms to satisfy different network level service level agreements (SLA)s on throughput and delays.

For cloud datacenter networks, our scheme uses a light weight front end server (FES) to allow the use of multiple NNS and there by mitigate the shortcomings of existing architectures. For hybrid P2P networks our schemes ensure high and accurate incentives to participating peers. Such fair incentives attract more peers which securely download and distribute contents. The thesis also presents efficient content index management schemes for the hybrid P2P networks with robust incentive implementation mechanisms.

We have implemented our protocols for general networks (the Internet), for cloud datacenter and hybrid P2P networks in the well known NS2 simulator. We have conducted detailed packet level and trace-based experiments. Simulation results show that our protocol for general networks can result in the reduction of the average file completion time (AFCT) by upto 30% when compared with well known existing schemes. Our cross-layer design for cloud datacenter networks can achieve a content transfer time which is about 50% lower than the existing schemes and a throughput which is higher than existing approaches by upto than 60%. Our detailed trace-based experiments also shows that our hybrid P2P protocol outperforms existing schemes in terms of file download time and throughput by up to 30% on average. The results also demonstrate that our hybrid P2P scheme obtains fair uplink prices for the uploaders and fair cost for the downloaders maintaining an overall system fairness. Besides, the results show the efficient enforcements of the prioritized allocations. Our implementation of the hybrid P2P protocol using an Apache SQL Server with PHP in Linux virtual machines demonstrates that content index management mechanisms of our protocol are scalable.

To all genuine, peaceful, kind and good people who are honestly striving for a truly better world for us all.

Acknowledgments

Ack will be here.

Table of Contents

List of Tables . . . . . **ix**

List of Figures . . . . . **x**

List of Abbreviations . . . . . **xii**

List of Symbols . . . . . **xiii**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Major Challenges of Distributed Systems 2
1.3 Existing Approaches 6
1.3.1 Single User Source Single Destination (SUSSD) / Single Source Single User Destination (SSSUD) Scenario 6
1.3.2 Single User Source Multiple Destinations (SUSMD) / Multiple Sources Single User Destination (MSSUD) Scenario 8
1.4 Our Proposed Approaches 10
1.4.1 SUSSD/SSSUD Case 10
1.4.2 SUSMD/MSSUD Case 11
1.4.3 Main Contributions 14
1.4.4 Research Statement 19
1.5 Thesis Structure 19

Chapter 2 *QCP:* **A Quick Cross-Layer Congestion Control Protocol** **20**
2.1 Introduction 20
2.2 QCP Algorithm 21
2.3 QCP Rate 22
2.4 QCP Can Achieve Processor Sharing (PS) 23
2.5 Efficient Sharing (ES) 24
2.5.1 ES vs PS and GPS 24
2.5.2 Single Bottleneck Scenario 26
2.5.3 Multi-Bottleneck Network 27
2.6 QCP Packet Header Format 28
2.7 Weighted QCP 29
2.8 Stability Analysis 30
2.8.1 Lyapunov Stability 30
2.9 Gradual Deployment of QCP 31
2.9.1 QCP Rate Using OpenFlow 31
2.9.2 Using Stateless OpenFlow 32
2.9.3 Using OpenFlow Edge Switches 32
2.9.4 QCP Rate Limiting 33
2.10 QCP with TCP Flows 33

2.11 Simulation Analysis . . . 34
2.12 Related Work . . . 38
2.12.1 TCP and its Variants . . . 38
2.12.2 Major Clean Slate Protocols . . . 39
2.13 Summary . . . 42

Chapter 3 *SCDA*: Cross-Layer SLA-aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval . . . 44
3.1 Introduction . . . 44
3.2 Network and Content Model . . . 45
3.2.1 Network Model . . . 45
3.2.2 Content Model . . . 46
3.3 SCDA Components . . . 46
3.3.1 Nodes . . . 46
3.3.2 Resource Monitors and Allocators . . . 47
3.4 SCDA Rate Metric . . . 48
3.4.1 Prioritized Rate Allocation for a Desired QoS level . . . 50
3.4.2 SCDA with OpenFlow . . . 51
3.4.3 QoS By Explicit Reservation . . . 51
3.5 SCDA Algorithm . . . 51
3.6 Global and h-Level Rate Allocation . . . 51
3.6.1 Obtaining the Rate values using Max/Min Algorithm . . . 52
3.7 Cloud Server Selection . . . 53
3.7.1 Interactive Content . . . 54
3.7.2 Semi-interactive Content . . . 54
3.7.3 Passive Content . . . 55
3.7.4 More Power Efficient Server Selection . . . 55
3.8 Serving Requests . . . 56
3.8.1 Serving External Write Request . . . 56
3.8.2 Serving Internal Write Request . . . 58
3.8.3 Serving External Read Request . . . 58
3.8.4 Updating Rate of On-going Flows . . . 60
3.9 General Network Topologies . . . 60
3.10 Experimental Results . . . 61
3.10.1 Using File Size and Flow Arrival Traces . . . 62
3.10.2 Using Pareto File Size and Poisson Flow Arrival Distributions . . . 66
3.11 Related Work . . . 67
3.12 Summary . . . 70

Chapter 4 *Hincent:* Cross-Layer Quick Content Distribution With Priorities and High Incentives . . . 71
4.1 Introduction . . . 71
4.2 *Hincent* Protocol . . . 73
4.2.1 Network and Content Model . . . 73
4.2.2 Logical and Physical Architectures . . . 74
4.2.3 *Hincent* Algorithms . . . 76
4.3 *Hincent* Rate and Price Calculation . . . 80
4.4 Content Source Selection . . . 82
4.4.1 Highest Rate to Price Ratio Policy (HRPR) . . . 82
4.5 *Hincent* is TCP friendly . . . 82
4.6 When a Flow Ends . . . 83
4.7 Other Server Selection Policies . . . 84
4.7.1 Highest Rate Policy . . . 84
4.7.2 Best Rate Fit Policy . . . 84

4.7.3 Smallest Price Policy . . . 84
4.7.4 Lowest Latency (Local Network) Policy . . . 84
4.7.5 Small Content Lifetime (Hop Count) Policy . . . 85
4.7.6 Private Group Policy . . . 86
4.8 Content Index Management . . . 86
4.8.1 Content Index Database . . . 86
4.9 Scaling User and Transaction Management . . . 91
4.9.1 Database Partition and Aggregate . . . 92
4.9.2 CID Complexity Analysis . . . 94
4.10 *Hincent* with Scarce Backbone Bandwidth . . . 94
4.11 *Hincent* Using Surrogate Servers . . . 95
4.12 Evaluation . . . 97
4.12.1 Simulation Setup . . . 97
4.12.2 Pure CDN Vs *Hincent*-Based Schemes . . . 98
4.12.3 Other P2P schemes Vs *Hincent* . . . 98
4.12.4 3D Streaming Result . . . 100
4.12.5 More Trace-Based Experiments . . . 100
4.12.6 CID Implementation Experiments . . . 106
4.13 Related Work . . . 109
4.14 Summary . . . 111

Chapter 5 Conclusion and Future Work . . . 112

References . . . 113

List of Tables

2.1 QCP Notations 22
2.2 Baseline parameters for experiments on estimation of N 34
2.3 QCP versus RCP under high load scenario: Poisson(8333.3 flows/sec), Pareto(1.2,30 pkts) . . 35
2.4 AFCT (seconds): QCP versus XCP 36

3.1 SCDA Parameters 48

4.1 *Hincent* Parameters 76
4.2 Peer Table Fields 88
4.3 Content Information Table Fields 89

List of Figures

1.1 Single User Source Single Destination (SUSSD) / Single Source Single User Destination (SSSUD) Scenario . . . 2
1.2 Datacenter: Single User Source Multiple Destination with Single Path (SUSMDSP) / Multiple Sources Single User Destination with Single (possible) Path (MSSUDSP) Scenario . . . 3
1.3 Datacenter: Single User Source Multiple Destination with Multiple Path (SUSMDMP) / Multiple Sources Single User Destination with Multiple (possible) Path (MSSUDMP) Scenario 4
1.4 Hybrid P2P: Single User Source Multiple Destination with Single Path (SUSMDSP) Scenario 5
1.5 Our Proposed Scheme: SUSSD/SSSUD Case . . . 11
1.6 Our Proposed Scheme: SUSMD/MSSUD Datacenter Case . . . 12
1.7 Our Proposed Scheme: SUSMD/MSSUD Hybrid P2P Case . . . 14
1.8 Thesis Dimensions . . . 15

2.1 QCP header with 12 bytes . . . 28
2.2 QCP header with 8 bytes . . . 29
2.3 Estimation of N versus time with $\alpha = 0.1$, $\alpha = 1$ for RCP . . . 35
2.4 AFCT versus number of flows . . . 36
2.5 AFCT of *Naive QCP* vs RCP: Poisson(1500 flows/sec), Pareto(1.2, 25pkts) . . . 37
2.6 FCT of flows versus simulation time (with 2 sec aggregation): Poisson(8333.3 flows/sec), Pareto(1.2, 30 pkts) . . . 38
2.7 FCT CDF of finished RCP flows vs QCP flows: Poisson(8333.3 flows/sec), Pareto (1.2, 30pkts) 39
2.8 FCT of XCP flows vs QCP flows: Poisson(6000 flows/sec) . . . 40
2.9 FCT of XCP flows vs QCP flows: Poisson(5400 to 30000 flows/sec) . . . 41
2.10 Two bottleneck network topology . . . 41
2.11 FCT of Group 0 (G0) and Group 1 (G1) RCP and QCP flows . . . 42
2.12 FCT CDF of RCP flows vs QCP flows: Flow inter-arrivals and sizes taken from traces . . . 43

3.1 SCDA Architecture . . . 47
3.2 The SCDA Max/Min Scheme . . . 54
3.3 Serving External Write Request . . . 57
3.4 Serving Internal Write Request . . . 58
3.5 Serving External Read Request . . . 59
3.6 The SCDA Max/Min Scheme With Multiple Possible Paths per Server . . . 61
3.7 RMs and RAs Used with General Network Topolgies . . . 62
3.8 SCDA experimental network topology: Propagation delay of each local datacenter link is 10 μs 62
3.9 Instantaneous throughput comparison of SCDA and RandTCP based: Using Video Traces with control flows . . . 63
3.10 FCT CDF comparison of SCDA and RandTCP based: Using Video Traces with control flows 64
3.11 AFCT comparison of SCDA and RandTCP based: Using Video Traces with control flows . . 64
3.12 Average Instantaneous Throughput comparison of SCDA and RandTCP based: Using Video Traces without control flows . . . 65

3.13 FCT CDF comparison of SCDA and RandTCP based: Using Video Traces without control flows . . . 65
3.14 AFCT comparison of SCDA and RandTCP based: Using Video Traces without control flows 66
3.15 AFCT comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 1$ . . . 66
3.16 FCT CDF comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 1$. 67
3.17 AFCT comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 3$ . . . 67
3.18 FCT CDF comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 3$. 68
3.19 Throughput comparison of SCDA and VL2: Using Pareto and Poisson Distributions 68
3.20 File Completion Time (FCT) comparison of SCDA and VL2: Using Pareto and Poisson Distributions . . . 69

4.1 The *Hincent* Architecture . . . 74
4.2 The *Hincent* Logical Architecture . . . 75
4.3 *Hincent* Logical Components . . . 76
4.4 The *Hincent* Architecture . . . 85
4.5 The CID Architecture . . . 87
4.6 Encrypted Content Transfer Using CIM . . . 89
4.7 The CID Partition and Aggregation . . . 93
4.8 *Hincent* with Surrogate Servers . . . 96
4.9 Pure CDN based Approach . . . 98
4.10 *Hincent*-based Approach . . . 99
4.11 Avg and Max CCT of *Hincent* Vs TCP-Based Approaches . . . 99
4.12 Avg Instantaneous Throughput Per Peer for Streams 1, 2 and 3 . . . 100
4.13 Avg Instantaneous Throughput Over Time for Streams 1, 2 and 3 . . . 101
4.14 File completion time with 1 YouTube server . . . 103
4.15 Average file completion time (AFCT) with 1 YouTube server (small files) . . . 103
4.16 Average file completion time (AFCT) with 10 YouTube server . . . 104
4.17 Net amount to pay in dollars per GB of downloaded content with 1 YouTube server . . . 104
4.18 Net amount to pay in dollars per GB of downloaded content with 10 YouTube servers (First few peers) . . . 105
4.19 Net amount to pay in dollars per GB of downloaded content with 10 YouTube servers (All peers) . . . 105
4.20 Query time using the tblSelectedSource table . . . 107
4.21 Query time using SQL JOIN from all tables . . . 107
4.22 Query time from peers to the CIM . . . 108

List of Abbreviations

List of abbreviations may be here.

List of Symbols

List of symbols may be here.

Chapter 1

Introduction

1.1 Motivation

Various types of distributed systems such as the Internet and other large and small scale networks have been shaping the way we live. The *Internet* which is the biggest distributed system and other types of distributed systems such as *cloud data center* networks and *peer-to-peer* have been growing so fast and complex. Our reliance on them has also been growing as fast.

The growth of such distributed systems translates into an explosive growth of online content [2, 3]. These online contents are generated either by centralized content providers (Comcast, Amazon, etc) or distributed users (Youtube, Facebook, gmail, etc). Such content generation is expected to grow even more (40-45% a year) [3] with the further expansion and sophistication of the Internet and networking technologies. This fast growth and complexity of these distributed systems brings along many challenges.

General Internet and data center content (traffic) is dominated by a large number of small flows (mice) and a few number of large flows (elephant) [4, 5, 6]. For certain small size time sensitive requests (flows) in applications such as financial and database transactions, even a small increase in average flow completion time (AFCT) is significant. Multimedia (streaming) applications also require a more smooth, fair and predictable rate allocation. Video streaming services such as YouTube and Netflix require an average download rate which is slightly larger than video encoding rate [7]. Otherwise, if elephant flows significantly delay the periodic transfer of fixed size blocks (64KB for instance) of video content [7], the video playback may interrupt due to empty buffers. So the AFCT of flows of specific sizes becomes an important performance metric as also discussed in [8]. To achieve small AFCT, high and smooth throughput for majority of network flows, *efficient congestion control and routing* protocols become a necessity.

Besides, users of online (multimedia) content have diverse Quality of Service (QoS) requirements. Based on their QoS specifications, content users make service level agreements (SLA) with content and/or network providers. Satisfying QoS requirements to all users with dynamic network and server loads and with limited resource capacities (link bandwidth, server storage, processing, energy) is challenging. The main challenge is

in finding the corresponding content transfer *rates* and *routes* for the user content flows. Such rate allocations and path (route) computations should *minimize content transfer time (AFCT)* and achieve high and smooth throughput by *utilizing* available resources. Finding such best rates and routes for user flows translates into *efficient congestion control and routing* schemes.

In rest of this chapter, we will first present the major *congestion control and routing* challenges of distributed systems in section 1.2. We will then discuss the existing schemes to address these challenges in section 1.3. In section 1.4, we introduce our proposed solutions to deal with each of the challenges. Finally, in section 1.5 we explain how the rest of this thesis is organized.

1.2 Major Challenges of Distributed Systems

Content transfer and communication in distributed systems such as the *Internet*, *datacenter networks* and *peer-to-peer (P2P) systems*, involve content sources and content destinations. The number of possible content sources/destinations can be just one or multiple nodes.

In a *general Internet* networks, there is usually a single web server which is the only possible source of a per user content as shown in figure 1.1. We call this scenario, a single user source single destination (SUSSD) and single source single user destination (SSSUD).

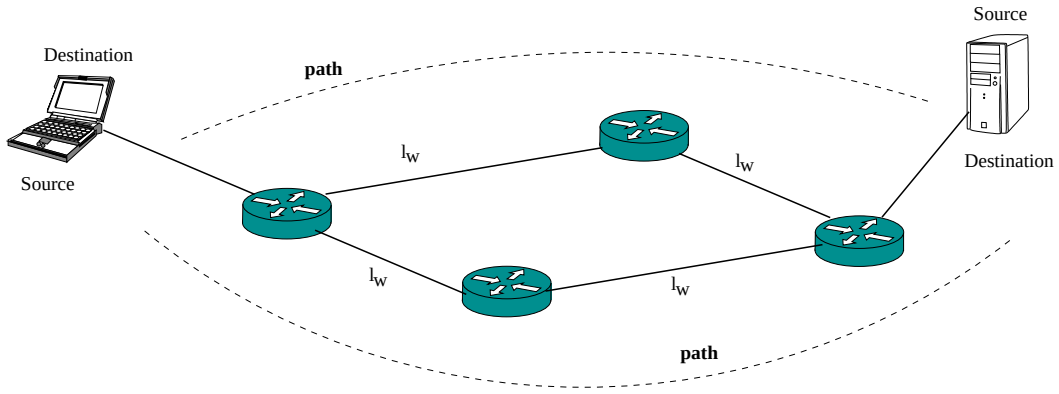

Figure 1.1: Single User Source Single Destination (SUSSD) / Single Source Single User Destination (SSSUD) Scenario

In *datacenter networks* a user can upload (store) content to any one of multiple possible servers as shown in figure 1.2 and 1.3. The content is usually replicated. The user can then download (retrieve) its content from any of the replica or original servers. We call this scenario a single user source multiple destinations with single (possible) path (SUSMDSP) and multiple source single user destination with single (possible) path (MSSUDSP). There may be only single possible path to/from each server as shown in figure 1.2. There may also be multiple possible paths to/from the server as shown in figure 1.3.

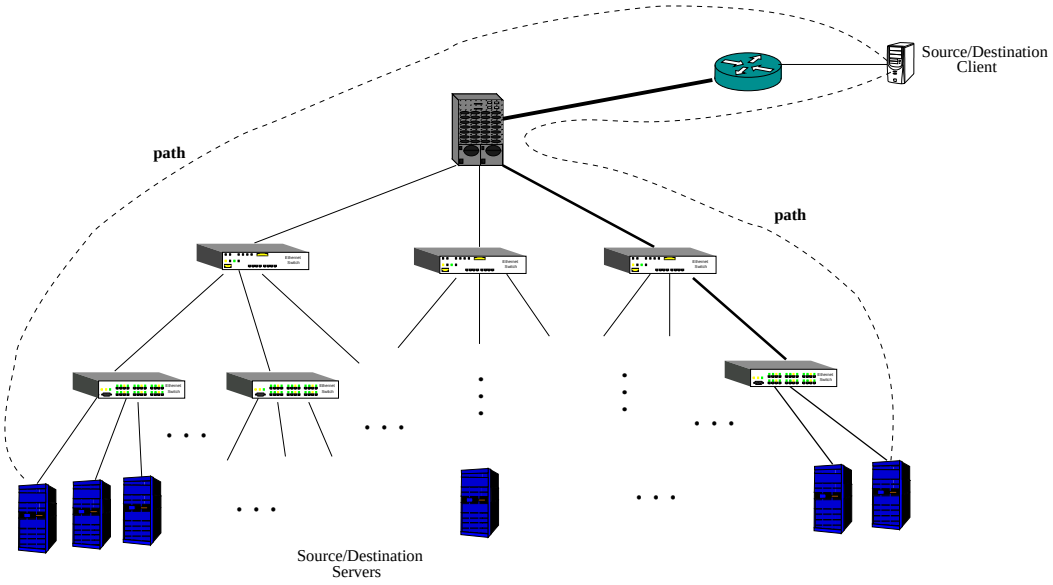

Figure 1.2: Datacenter: Single User Source Multiple Destination with Single Path (SUSMDSP) / Multiple Sources Single User Destination with Single (possible) Path (MSSUDSP) Scenario

Similarly in *hybrid P2P networks*, the same content may exist at a content distribution network servers (CDN) or at multiple user nodes (peers) resulting in multiple possible content sources, one of which is selected at a time as shown in figure 1.4. In pure P2P networks, content is exchanged among peers with no centralized source feeding the CDN. However in the hybrid case, a content source can be either the CDN or any of the user peers that has the content. As can be seen from figure 1.4, the bottleneck link is usually the last mile link connecting the peers to the Internet backbone network. The backbone links represented by the "Internet" node in figure 1.4 are not usually congested as can also be seen from [9]. A network service level agreement a user can have with its network service providers can also help ensure the desired link capacity. Besides, a peer downloading/uploading contents from/to multiple sources/destinations makes the last mile link the most likely bottleneck. Most of the work regarding *hybrid P2P networks* in this paper makes this assumption. However, our design presents schemes to deal with scenarios where the bottleneck link can be anywhere in the backbone.

Multiple possible content sources/destinations means many possible paths to a content source/destination as shown in figures 1.2,1.3 and 1.4. Hence selecting a content source/destination translates to finding a path to a content source/destination and thus routing. In the rest of the thesis a content source/destination is called server and server selection has equivalent meaning as routing. This is because a specific server selection results in a specific path (route) computation to/from content source/destination (the server).

To transfer a content from a source to a destination in both single or multiple possible source/destination scenarios, distributed systems need to find the best *route* from content source to content destination and the

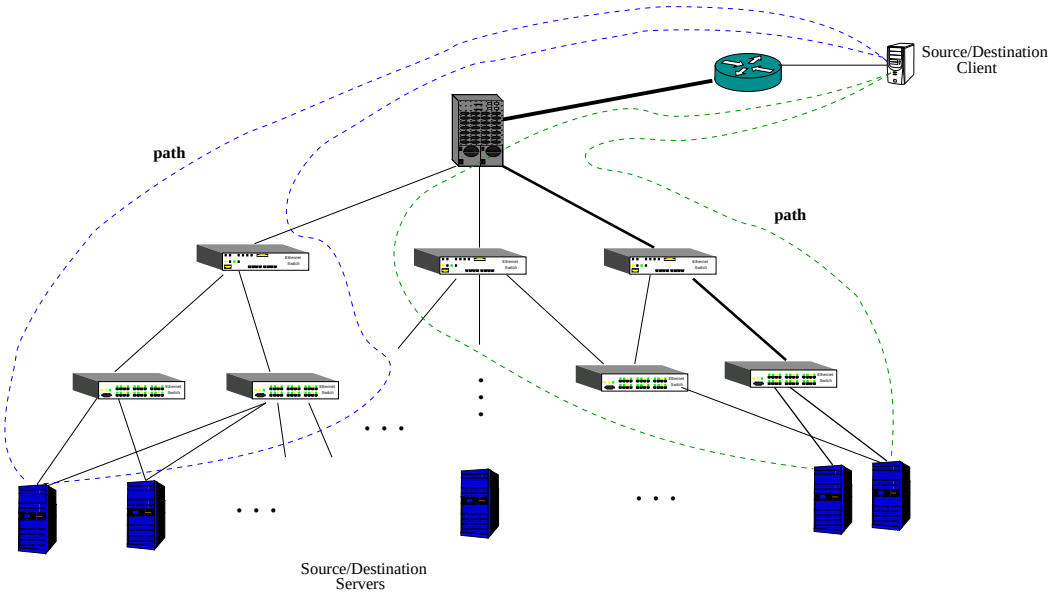

Figure 1.3: Datacenter: Single User Source Multiple Destination with Multiple Path (SUSMDMP) / Multiple Sources Single User Destination with Multiple (possible) Path (MSSUDMP) Scenario

rate at which content is transfered. Finding the best route and best transfer rates translates into the best congestion control and efficient routing of contents. Hence the design of distributed systems should address the unified research problem of finding the optimal **route** and flow sending **rate** that **minimize content transfer time** and achieve **high and smooth throughput** with **efficient resource usage**. The main challenges/constraints that arise in addressing this research problem are the following.

- **Network-Wide Max/Min Fairness:** How can such rate and route be computed in such a way that max/min fairness is ensured where available resource is fully utilized as long as there is demand for it?
- **Network-Wide SLA:** How can *SLA violation* be detected in realtime (milliseconds interval) and be mitigated in the process of rate and route computation at a network level?
- **Less Architectural Changes:** How can such data transfer rate allocations to different users and routing or server selection be enforced with minimal (less) or no changes to routers, switches and the TCP/IP stack?
- **QoS:** How can prioritized resource allocation be made to different users and content flows in realtime and with small overhead. Such allocation can for instance allow some applications such as multi-view 3D streaming to assign higher rate to some flows (streams)?
- **Incentives:** For MSSUD distributed systems such as hybrid P2P where any peer can be a content source, what is the most cost-effective and efficient incentive for peers for offering their resources (link

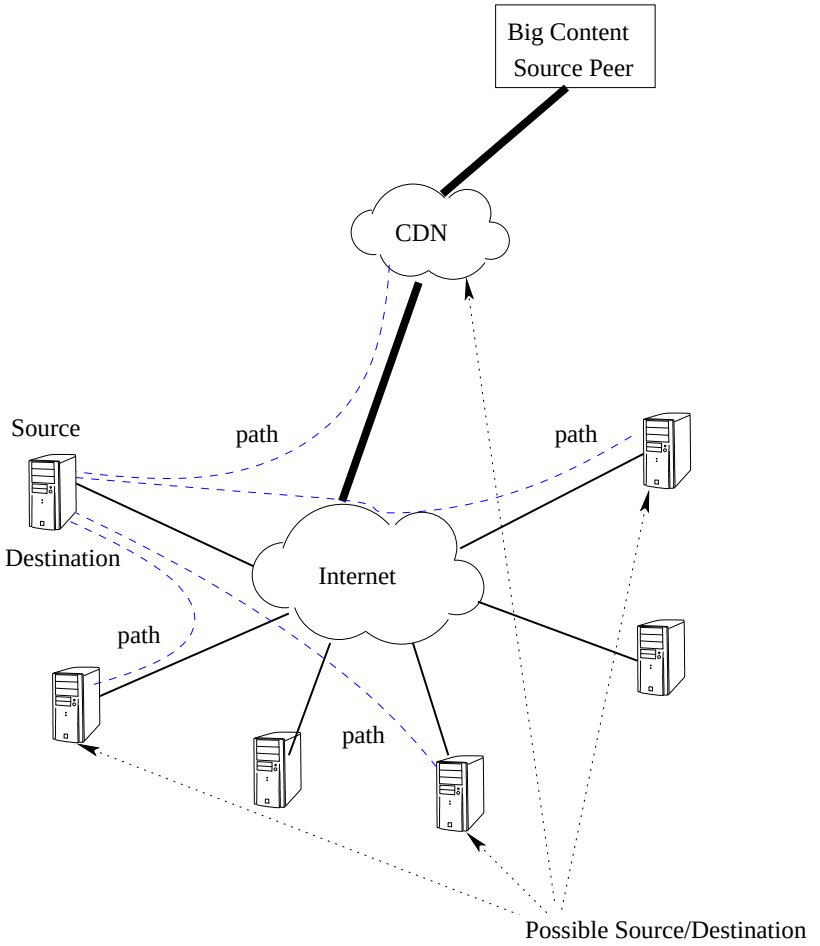

Figure 1.4: Hybrid P2P: Single User Source Multiple Destination with Single Path (SUSMDSP) Scenario

bandwidth, storage, etc)?

- **Metadata and Content Index Management:** Cloud datacenter networks and hybrid P2P systems need to manage metadata information such as which content is stored in which server. So how best can metadata and content index be managed?

- **Surrogate Servers:** If peer nodes are not reliable enough, emerging cloud or cloudlet [10, 11] technologies can be leveraged to exchange content. How can such surrogate servers be used to achieve small content transfer times and high smooth throughput?

- **Secure Content Exchange:** How can content be securely transmitted to an accountable peer in hybrid P2P networks?

- **Power Efficiency:** How can the server selection be made in a power-efficient manner in the case of SUSMD/MSSUD for datacenter networks.

1.3 Existing Approaches

In this section we discuss existing schemes to address the congestion control and routing challenges discussed in section 1.2 above.

1.3.1 Single User Source Single Destination (SUSSD) / Single Source Single User Destination (SSSUD) Scenario

In this section we will discuss how existing literature approaches the congestion control and routing problems for the SUSSD/SSSUD scenario and their drawbacks.

Congestion Control

The majority of network traffic uses the transmission control protocol (TCP) [12] as a congestion control protocol. TCP was successful managing congestion in the early stages of the Internet and before the emergence and vast expansion of other types of network and networking technologies. With the growth of Internet and network technologies TCP encountered many performance problems [13, 14]. As discussed in [14], a random packet loss can for instance result in a significant TCP throughput degradation over high bandwidth-delay product networks. TCP is also not fair to short flows (mice) which are the majority of network traffic as few big size (elephant) flows can significantly delay many mice flows [4, 6, 15].

There have been numerous research efforts to deal with the weaknesses of TCP. The current modifications to TCP such as HighSpeed TCP [16] inherit the main problems of TCP in not quickly knowing the bottleneck link share of flows. This results in flows taking longer to finish than necessary [8].

The eXplicit congestion Control Protocol (XCP) [17] is designed to achieve full link utilization and hence high per flow throughput. However, XCP is not fair to short flows (mice) resulting in higher average flow completion time (AFCT) [8]. The Rate Control Protocol (RCP) [8] on the other hand was designed to finish flows quickly. But as shown later in this paper, RCP under or over estimates the number of active flows which it needs to obtain the rate at which flows send packets. This results in under or over utilization of bottleneck link capacity which in turn results in high queue length, packet drops and high AFCT. More related work discussion is presented in section 2.12.

Routing

The Open Shortest Path First (OSPF) [1] is the currently deployed routing protocol to find a path from one node in a local autonomous network to another node (entity). It is the most commonly used intra-domain

Internet routing protocol. However OSPF as well is finding it increasingly difficult to cope with the growth in size and complexity of distributed systems. One of the main problems with the existing such Shortest Path Routing (SPR) is the simple heuristic routing metric (link weight) they use. The routing metric used doesn't properly take into account the latest status of the network.

Lack of efficient routing and congestion control protocols and algorithms has been forcing owners of big distributed systems to over-provision their resources (networks). Unfortunately apart from the cost of upgrading the network (distributed system), the Moores Law-like technology over-provisioning trend with the growth of for instance the Internet is not sufficient to contain congestion as shown by Akella et. al [18]. This is because, as the authors pointed out, the maximum congestion in the Internet scales poorly with the growing size of the Internet graph. Akella et.al have further shown that the famous SPR which is the routing protocol in the Internet today can be worse than the Border Gateway Protocol (BGP) which is a Policy Routing. This surprising result is not because trial and error is better than a scientific approach. It only exposes with a counter example the weaknesses of the existing SPR protocol demanding for a more clever and comprehensive scientific approach, something we hope to deliver in this work.

A traffic engineering technique based on some ideas of XCP, (TeXCP) [19] was also proposed to address some routing issues. But TeXCP also inherits some of the unfairness properties of XCP. Besides, an Internet Service Provider (ISP) needs to configure each TeXCP ingress-egress agent with a set of K-shortest paths it can use to deliver its ingress-egress (IE) traffic. These paths are then pinned using a standard protocol like Multi-protocol Label Switching (MPLS) [20]. The shortest path here doesn't take congestion into account. It merely uses propagation delay as a link metric. So essentially if all these links are congested TeXCP will stick with them even if there are other less congested paths. TeXCP also needs additional probe packets to discover the utilization in each path. A traffic engineering (TE) technique for MPLS networks [21] was also proposed. But it is based on the notion that the number of flows (LSP requests) through a link can be known and is hence difficult to apply for non MPLS networks. Wang et.al [22] proposed an edge-based TE for OSPF networks. The scheme called a *k-set* TE method, partitions traffic into uneven k traffic sets at the edge of a network. For each traffic class (set), the k-set approach uses residual bandwidth (spare) capacity as a link weight and relies on a heuristic to solve a mathematical programming formulation. Such spare bandwidth link weight doesn't take into account the number of active flows in each link. For instance two links with the same spare bandwidth but different number of active flows are treated the same way. But this is not always true as the link with more flows is highly likely to be more loaded with time.

1.3.2 Single User Source Multiple Destinations (SUSMD) / Multiple Sources Single User Destination (MSSUD) Scenario

In this section we discuss how existing approaches address the routing and congestion control challenges with multiple sources/destinations.

Cloud Datacenter

Existing attempts to solve the server (content source/destination) selection problem broadly fall into two categories. The first one is using large content distribution networks (CDNs) such as Akamai. Such CDNs use a large number of edge servers distributed in vast Internet locations. As explained in [23, 24] such schemes select a server for client request based on proximity and latency. Server selection is not based on best content transfer rates and lowest delays. Besides, the scalability of distribution and maintenance of edge-servers scattered in many Internet locations all over the world is costly. The work presented in [24] shows that significant consolidation of Akamai's platform into fewer large datacenters is possible without degrading performance.

The second but dominant and emerging content storage and retrieval approach is using cloud data centers. There have been numerous data center architectures [25, 26] to address the above-mentioned challenges. However such architectures do not use an efficient mechanism to select the best servers in the data center. They use random switch (server) selection strategies. They also rely on the transmission control protocol (TCP) [27] to control the rates of the senders. TCP is known to have higher average file completion time (AFCT) than necessary as discussed in [8]. Besides, such approaches are restricted to specific structure of datacenter network interconnect.

The Google File System (GFS) [28] and its derivatives such as the Hadoop File System (HDFS) [29] which are the most commonly used distributed file systems for cloud computing are designed to meet these challenges. These cloud systems use a single name node server (NNS) to keep metadata, replication and location information of all data chunks stored in the cloud. With the growth of cloud network demand, the single name node can become a bottleneck resource and a single point of failure. Besides, such cloud systems use TCP with all its weaknesses to transfer data from one node to another node. They also do not have an efficient scheme to decide where to store data, where to retrieve it from and how to route it. There has been a modification of TCP for data center networks (DCTCP) [30] an effort to improve on TCP for data center networks. However, DCTCP also inherits the problems of TCP that it depends on packet loss and packet markings as congestion signal. Besides DCTCP makes an unrealistic assumption in the derivation of its main threshold parameter. For instance it assumes that flows are synchronized following identical

congestion avoidance sawtooth (no slow start) and with the same RTT. The recommended queue threshold parameters are ranges and not specific values making it difficult to decide what value to choose. The simulation setup used to validate DCTCP was also too simplistic to show the effects of these assumptions. Moreover, DCTCP trades off convergence time; the time required for a new flow to grab its share of the bandwidth from an existing flow with a large window size. They argue this is OK as DCTCP is designed for short RTT networks. So DCTCP cannot for instance be a good fit for scenarios where a main cloud controls the communication to cloudlets or customer networks at a considerable distance. More related work discussion is given in section 3.11.

Hybrid P2P

Traditionally, centralized content providers (CCP) use content distribution networks (CDN) to distribute their contents to their customers. With the increase in high bandwidth content demands [2], content providers should either over-provision their bandwidth to handle peak demands or rely on purchased service such as Akamai. However as discussed in [31] it is cheaper for content providers to purchase bandwidth from their users than using third party content distribution networks (CDNs) or purchasing the infrastructure to directly serve contents. Besides assisting CDNs, using P2P networks results in significant scalability gains as discussed in [32, 33].

While using cooperative customer peers to distribute content, providers need to be mindful about incentives to pay back peers for their upload bandwidth. Besides, content providers need to make sure that the incentives and returns are accurate enough to offer better quality of service (QoS) guarantees. Using an efficient, fair and accurate peer incentive mechanism can also benefit content providers and network operators significantly. Content providers can save on bandwidth cost by buying peer link bandwidth. Besides, peers who get significant credit (financial or content credit) from uploading content are most likely to subscribe to more contents potentially increasing the content demand. More content demand can also translate into more link bandwidth demand which can benefit network operators. As discussed in [34], distributed user-generated contents can also be feasibly shared from homes allowing users (peers) full ownership and control of their contents.

Existing incentive-based content sharing mechanisms such as Price-Assisted Content Exchange (PACE) [35, 36] and Dandelion [31, 37] do not use efficient incentive mechanisms. For instance, PACE does not guarantee fair-exchange of content for payment. Dandelion uses fixed bandwidth pricing mechanism that peers do not decrease their prices to attract more customers when they have high upload rate and vice-versa. Besides, such existing schemes do not find and enforce accurate rates at which peers can download content from other

peers so as to minimize content transfer time. They do not give a mechanism to prioritize content transfers which is an important component of 3D [38] and other multi-view streaming applications where some streams are more important than others based on the view angle. Besides, existing work does not provide an efficient content source selection mechanism which chooses a source that leads to high throughput and low file completion time. More related work discussion is given in section 4.13.

As discussed above, all existing works approach the challenges discussed in section 1.2 separately coming up with a congestion control protocol and a routing protocol each of which are independent.

1.4 Our Proposed Approaches

To address the challenges in congestion control and routing discussed in section 1.2 above and to solve the drawbacks of existing schemes discussed in section 1.3 above, we design *cross-layer routing and congestion control* schemes. Our schemes derive a simple and effective congestion control and routing metric. The metric serves as a rate at which sources send data (avoiding congestion) and also as a link metric of a *max/min routing* scheme [39, 40, 41, 42]. The congestion control metric we use at the transport layer crosses a layer serving as a routing metric in the network layer. This makes our schemes cross-layer.

The cross-layer approach is demonstrated by simplified steps in figures 1.5, 1.6 and 1.7 for general, cloud datacenter and hybrid P2P networks respectively. First a per link $R_i(t)$ is obtained using efficient formulas discussed in chapters 2, 3 and 4. The schemes find the bottleneck link rate $R_{path\,i}$ of each possible path which is the minimum of the rates of the links of the corresponding path. The path with the maximum bottleneck rate R_{path} is then selected for the content transfer. The source also sets its sending rate to be R_{path}. This is how rate metric R_{path} is used as a link weight metric in a max/min routing algorithm described [39].

The way the path and rate are computed and the actual protocols involved for general networks, for cloud data center networks and for hybrid P2P networks are different. We exploit the topological structures and nature of content source/destination of cloud datacenter and hybrid P2P networks to formulate the corresponding protocols and algorithms. These algorithms require minimal involvement of routers/switches and the TCP/IP stack. We next discuss our cross-layer approach for the SUSSD/SSSUD case in *general networks* and for the SUSMD/MSSUD cases in *cloud datacenter networks* and *hybrid P2P networks*.

1.4.1 SUSSD/SSSUD Case

In general networks with no specific topological structure first, a fair rate is calculated for each link (interface). A path for each flow (content transfer) is then computed by treating the rates as link weights as shown in

figure 1.5. To set the sending rates of flows to the bottleneck rate of the selected path, a shim header is used between the TCP and IP headers of data and acknowledgement (ack) packets. The layer bottleneck rate is written to a field in the shim header of data packets and then copied back to the ack packets. The source then sets its congestion window to the product of the rate it gets from the ack packets and its round trip time (RTT).

The path is computed using a max/min routing algorithm such as the one in [39] either by OpenFlow controller or in a distributed fashion. The OpenFlow controller collects and uses the link rate values as link states to perform path computation on the behalf of the routers/switches. A modified Dijkstra algorithm can also be used to compute the path by a controller or in distributed manner. The modified Dijkstra uses the inverse of the link rate values as link weights. It then first computes the maximum such inverse value of each path and selects the path with the minimum such value. This translates into the maximum bottleneck rate value.

In the rest of the thesis we call such a cross-layer approach for general networks a *Quick Congestion control Protocol (QCP)*. We discuss QCP in great detail in Chapter 2.

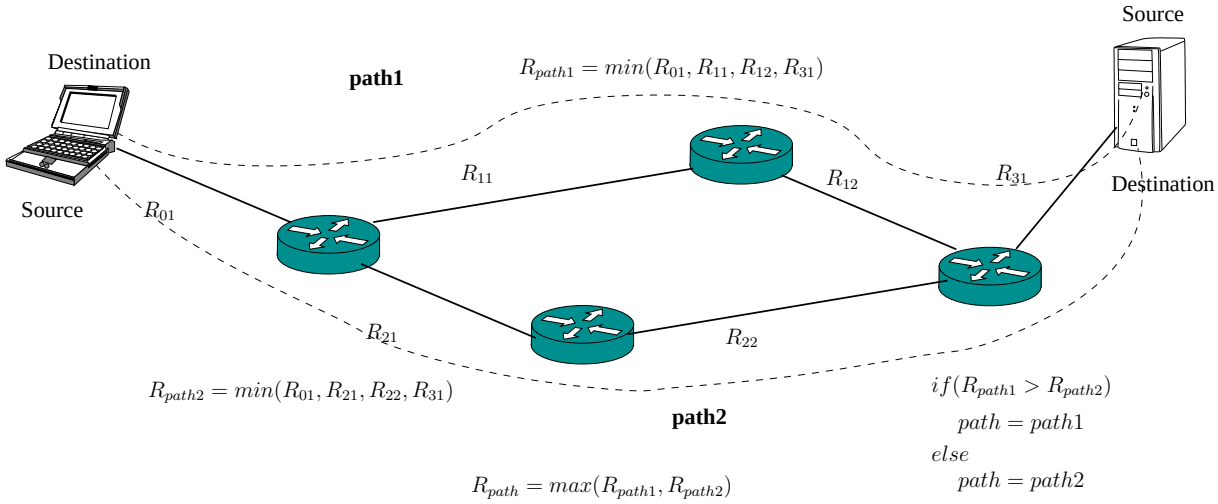

Figure 1.5: Our Proposed Scheme: SUSSD/SSSUD Case

1.4.2 SUSMD/MSSUD Case

In this section we will discuss the cross-layer approaches for the SUSMD/MSSUD case in datacenter and hybrid P2P networks.

Cloud Datacenter Networks

For cloud datacenter networks with a tree network topologies as shown in figure 1.3, we use software agents called resource monitors (RM) and resource allocators (RA) to perform our cross-layer approach. Each RM and RA are associated with a content source/destination and the switches/routers respectively. Logically, the RMs are at the bottom (leaf) of the tree while the RAs are at each higher level of the tree. Physically, all these software components can be consolidated in a few powerful servers. The lowest level RAs aggregate traffic load information from their children RMs to calculate content transfer rates for each link (interface) on the behalf of switches/routers and to find the best path/server. Similarly each level RA aggregates traffic load information from its children RAs to do the same. Each level RA maintains the best path/server in its level and the rate to/from the server. A datacenter admin software/server which we call name node server (NNS) can then consult an RA at a particular level, to select the best path and hence server for a given request. The NNS also informs the RM of the requesting node to set its sending rate to be the path rate.

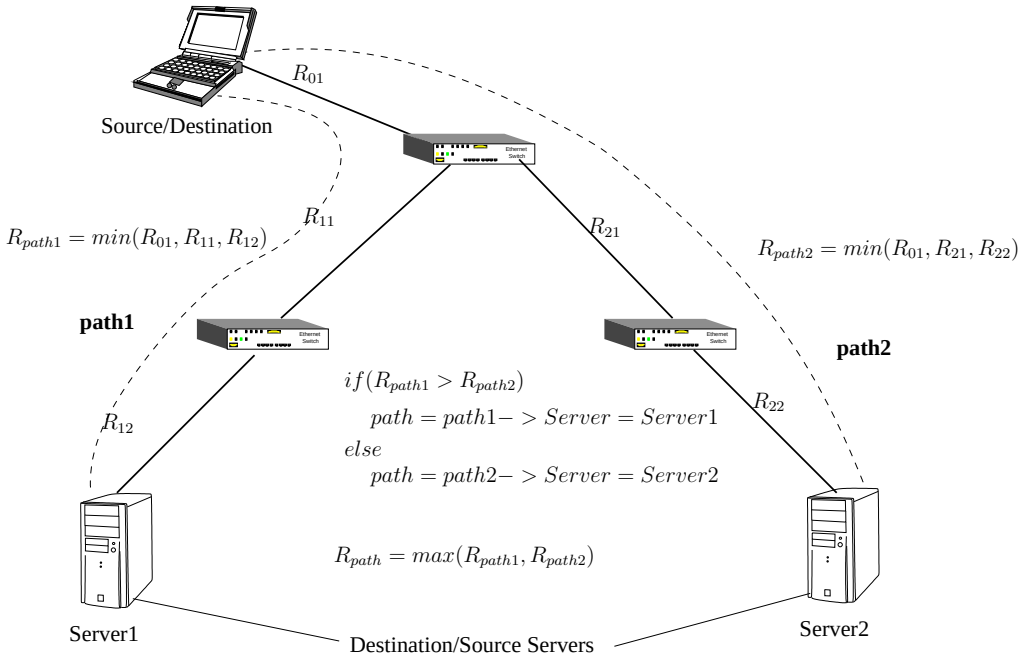

Figure 1.6: Our Proposed Scheme: SUSMD/MSSUD Datacenter Case

The RM and RA based scheme detects SLA violations and helps to automatically mitigate it in realtime. We call such a mechanism of finding a path and rate for datacenter networks SLA-aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval (SCDA). The SCDA protocol and associated algorithms are discussed in Chapter 3 in detail.

Hybrid P2P Networks

For hybrid P2P networks a content can be obtained either from a CDN source or from distributed peers that have the content as shown in figure 1.4. Similar to the RM of SCDA discussed in section 1.4.2 above, our hybrid P2P approach has a software agent called *peer agent* associated with each content source/destination. The PA interacts with another software component called content index manager (CIM) the way the RM of SCDA interacts with its RA. The PA and CIM cooperatively compute the paths and rates between peers. The CIM then selects a path and content transfer rate for a requesting peer.

There is similarity between cloud datacenter and hybrid P2P networks in selecting a content source/server for a requesting peer using RA and CIM respectively. In datacenter networks any of the replication servers can be a potential content source. Similarly, in hybrid P2P networks any of the peers which have the requested content can be a potential content source. In each case, the selected content source/path offers the highest throughput and minimum content transfer time. However, the possible set of content sources (replication servers) in the case of cloud datacenter networks is limited to a few of them by the NNS. The replication servers are usually selected when a content is written by the requesting user to the first cloud server.

On the other hand, in hybrid P2P networks there is no control and limit on the potential number of possible content sources. Any of one of the many nodes can be a content source. Hence CIM is required to manage the content. The software agent which maintains the list of replication servers for each user can also be considered as a CIM with a fixed number of replication servers per user. Besides, our hybrid P2P architecture does not have to select a destination server for a requesting user while our SCDA scheme selects a destination server for a content storage request. In the case of cloud datacenter networks, users are assumed to pay the cloud datacenter network service providers. Hence the RAs of SCDA do not have to explicitly deal with incentive mechanisms. In hybrid P2P networks, distributed peers can be content sources offering their resources to help with content transfer. Hence our hybrid P2P approach deals with incentive mechanisms. Our hybrid P2P cross-layer approach is called *Hincent*, Quick Content Distribution With Priorities and High Incentives in the rest of this thesis. Among other things, *Hincent* deals with secure content transfer using CIM as any of the participating peers can be potentially malicious. The details of our *Hincent* protocol and related algorithms are discussed in Chapter 4.

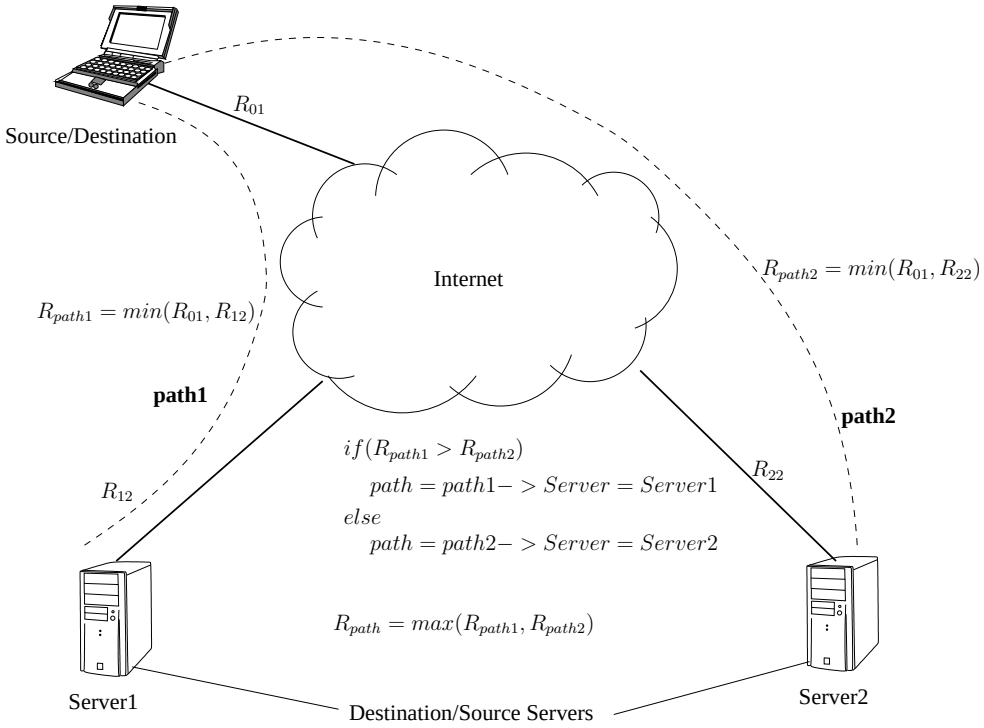

Figure 1.7: Our Proposed Scheme: SUSMD/MSSUD Hybrid P2P Case

1.4.3 Main Contributions

As described in figure 1.8 [1], our cross-layer approach is rate centric. For the distributed systems presented in this thesis, first (proportional/weighted) fair metric is derived. The rate metric is used as a link weight to find optimal routes (routing) and to determine the content transfer rates (congestion control). The rate based congestion control and routing scheme also serves to balance load among multiple resources (links, servers).

Different architectures are designed for general networks, datacenter and hybrid P2P networks to use the rate metric as a congestion control and routing metric as shown in figure 1.8. For instance one way of implementing the cross-layer scheme for general networks is by using a shim packet header with a few fields between the TCP and IP headers. We exploit the structure of datacenter and hybrid P2P networks to design our cross-layer scheme without the need of changing the TCP/IP packet headers and the modules of routers. To do this we use software components such as RM, RA, FES, NNS, etc for datacenter networks and CIM, PA, etc for hybrid P2P networks.

In our cross-layer schemes, the rate metric impacts numerous parameters to achieve SLA, QOS, guarantees and to provide incentives to peers in the case of hybrid P2P networks. To achieve these, the throughput

[1] Thanks Professor Klara Nahrstedt for the design of this figure.

and content transfer delay of each flow are functions of the rate. In the case of hybrid P2P networks, the per resource price needed to give incentives to participating peers is also a function of the rate. The higher the per flow rate, the lower the price (to attract more customers). The server power consumption by cloud datacenter servers is also impacted by the rate as the rate metric is used to move requests away from less power-efficient servers.

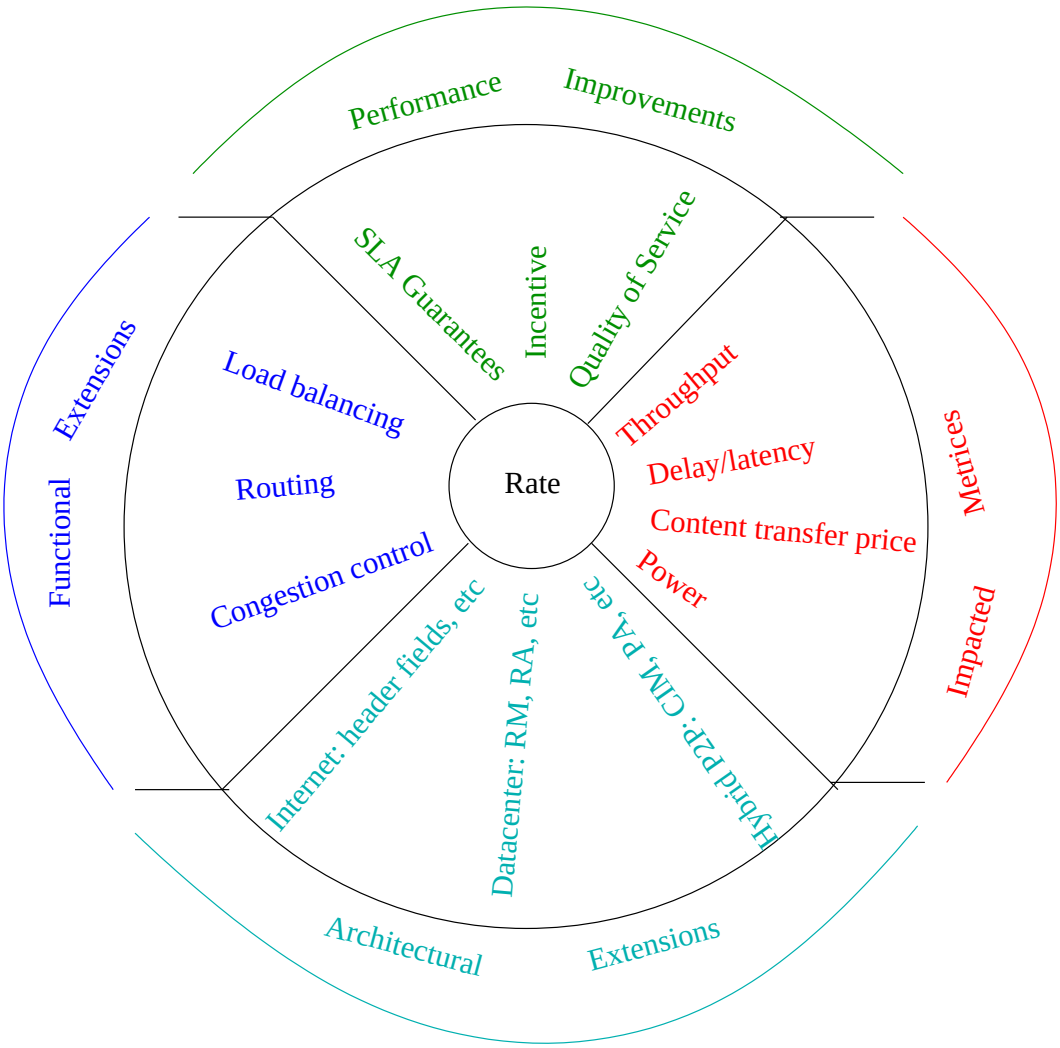

Figure 1.8: Thesis Dimensions

The main contributions of the thesis with such rate centric cross-layer approach, can be summarized as follows.

Efficient Cross-Layer Routing and Congestion Control Scheme

Efficient cross-layer routing and congestion control architectures for major distributed systems are presented. The distributed systems considered in this thesis are general networks, cloud datacenter networks and hybrid P2P networks.

Our cross-layer approach derives a simple and effective congestion control rate metric which is used as a link weight to compute path for content transfer. Content sources send data at this rate. Unlike TCP, this rate metric can quickly obtain a very high link utilization and a low queue size and hence results in smaller AFCT. Unlike XCP [17] the congestion control component of our approach is fair all flows resulting in smaller AFCT to majority of Internet and datacenter flows. It also uses an accurate derivation of the number of active flows and hence doesn't suffer from such estimation errors of RCP [8].

Path computation using our rate metric as a link weight results in selecting less congested paths when compared to existing schemes discussed in section 1.3 above. Using this cross-layer metric results in content transfer time smaller by upto 30% to 50% than existing schemes discussed in section 1.3 as shown in our experimental results. Our experimental results also show that our approach can result in throughput which is higher than existing approaches by upto than 60%.

Achieving Max/Min Fairness in Realtime

Our rate computation mechanisms can achieve network level max/min fairness in realtime (milliseconds range). The max/min fairness enables available resources to be utilized as long as there is demand for it. To achieve max/min fairness at every network resource (link), some content transfer flows are counted as fractional (partial) flows. For instance, let's consider two flows. One flow is counted as 0.5 because it cannot utilize the resource as a full flow. The second flow can fully utilize available resource is counted as 1 full flow. Then the resource capacity C is shared by 1.5 flows instead of 2 flows. This gives the second full flow $C/1.5$ of the capacity instead of $C/2$ of the flow. Existing weighted fair queuing or weighted processor sharing based schemes cannot achieve network level max/min fairness in realtime (milliseconds interval). Our adaptive dynamic cross-layer rate and path computations schemes allow us to achieve network level prioritized max/min fairness in realtime. The subsequent chapters of this thesis will discuss how our schemes achieve this max/min fairness in detail.

Minimal Overhead and Realtime QoS Mechanisms

Our proposed solutions can also achieve a desired quality of service (QoS) value by allocating different rates to different flows without causing router buffer overflow or link under-utilization. This is done by

using different priority weights of flows. Unlike existing QoS schemes (IntServ), our approach scales well as it does not hold per flow states. It also does not need such QoS support from routers and switches via explicit reservation which incurs extra overhead. Such existing explicit reservation techniques can also waste resources when reserved resources are not used. Our approach performs prioritized QoS allocations at packet and even byte level and at RTT time scales with max/min support. Besides, unlike existing QoS schemes, our approach uses accurate feedback of network bottleneck capacity to adjust the sending rates of flows, fully utilizing link capacities and avoiding congestion.

Detecting and Mitigating SLA Violations in Realtime

As part of our cross-layer rate computation, each source specifies its desired rate either inband using the data packets using a shim header or using control packets of our software agents. This design inherently allows us to detect violations in service level agreements (SLA) on content transfer rates at network level. If either a packet level per flow desired rate or the sum of per flow desired rates exceed the capacity of the local resource (link), then that is an indication of an SLA violation in realtime. An SLA violation signal can then trigger a backup path using non-congested links. Existing schemes pre-assign (reserve) maximum required capacities to maintain SLA. Such schemes can result in over-subscription and hence resource wastage. Our realtime packet level detection method does not compromise link (resource) utilization.

Efficient Incentive Mechanisms for Hybrid P2P Networks

The effectiveness of hybrid P2P networks depends highly on how efficient and fair the incentive mechanism for resource usage is. Our cross-layer approach for hybrid P2P networks offers per packet incentives to peers as a function of the rate of using their resources. Our design is more efficient than existing schemes discussed in section 1.3 above.

Less or No Change to Routers/Switches and TCP/IP Stack

The protocols proposed in this thesis either do not require changes to existing routers/switches or can be implemented with minimal (less) changes. The QCP algorithms can be easily implemented using (extending) the emerging OpenFlow [43, 44, 45] architecture. The designs of *SCDA* and *Hincent* use distributed software components which only read queue size information from router/switch interfaces using existing switch/router features. This enables *SCDA* and *Hincent* to be implemented without any change to existing router/switch architectures and the TCP/IP stack. Moreover, *SCDA* and *Hincent* do not need additional header in or changes to data packets and the TCP/IP stack. The software components communicate with

each other either using message passing, inter-process communication or small size control packets. The software components can then modify the congestion window size (*cwnd*) of the active TCP flows.

Power Efficient Server/Path Selection in Datacenter Networks

Our SCDA protocol for datacenter networks performs server selection by classifying contents into active and passive. This classification can be done using user specified parameters. The frequency of writes to and reads from the servers can also be used to classify a content as passive or active. Passive contents are then replicated or moved to servers with an uplink rate of greater than some threshold. The higher the uplink rate the more idle the server is. In datacenter networks, there is heterogeneity in power consumptions by different servers. This can be due to server's location in a rack, server age and specifications or due to other compute intensive or background tasks the servers perform. Our SCDA design can also perform more *power aware server selection* using a adaptive mechanism. To do this, SCDA relies on measurements of each server's energy consumption by (heat or temperature) sensors in the servers. The RAs at different levels of SCDA then select a server with the *highest rate to power ratio*.

Efficient Metadata Management For Datacenter Networks

Existing cloud-computing architectures (file systems) such as GFS and HDFS [28, 29] use only a single NNS (name node server) to manage the datacenter metadata. This can be a potential bottleneck resource and a single point of failure. To address this problem, SCDA allows multiple name node servers (NNS) using a light weight front-end server (FES) which forwards requests to the name nodes (NNS).

Efficient Content Index Management Schemes for Hybrid P2P Networks

Some peers of the hybrid P2P networks have contents to share and others request for a content. A *Hincent* CIM dynamically manages content requests and server selection. CIM selects a server based on a chosen policy. For instance CIM uses highest rate to price ratio policy where a peer with which offers the highest upload rate to price ratio is selected. The CIM stores registered content information (content metadata) at the specifically designed content index relational database (CID) which is part of the CIM system. Unlike existing schemes, the CIM uses map/reduce like database partition and aggregate steps with multiple CIDs. *Hincent* CIM Implementation using an Apache SQL Server with PHP in Linux virtual machines demonstrates that the CIM mechanisms are scalable.

Securely Exchanging Content in Hybrid P2P Networks

The CIM maintains secret key for each content it obtains from the content source. After a destination peer downloads the encrypted content from a selected source using a path which is almost always unknown to CIM, it asks the CIM for the key to decrypt the content. If the destination peer has enough credit, the CIM sends the secret key which the destination peer uses to open the content. The CIM also maintains content hash to ensure content integrity. Here it must be known that the CIM has no access to the actual data exchanged between peers. Hence data exchanged between the peers is not compromised.

Using Surrogate Servers to Exchange Content

When peer nodes do not have reliable or enough resources to share content with others, they can use a network of geographically distributed surrogate servers. In this thesis we also extend our *Hincent* cross-layer routing and congestion control scheme to the surrogate server network to achieve small content transfer times and high smooth throughput. These surrogate servers may be cloudlet [10, 11] or cloud servers. These servers have OpenFlow vSwitches (switches/routers). The servers may be linked with dedicated tunnels or overlay links over the Internet. Similar to QCP, SCDA and *Hincent*, the per flow rates are the link weights.

1.4.4 Research Statement

In light of the challenges and our proposed solutions discussed above, the research statement of this thesis reads as follows:

There exist cross-layer routing and congestion control schemes for distributed systems using a metric which offers a fair share to flows resulting in minimal average file transfer time and high throughput. The schemes can be implemented with minimal changes to existing network infrastructure.

1.5 Thesis Structure

The rest of the thesis is organized as follows. In Chapter 2 we present our cross-layer QCP protocol for general networks (the Internet). In Chapter 3, we discuss our SCDA cross-layer protocol for cloud datacenter networks in detail. We then present our *Hincent* protocol which is a cross-layer scheme for hybrid P2P networks. Finally we give a summary of the thesis in Chapter 5.

Chapter 2

QCP: A Quick Cross-Layer Congestion Control Protocol

2.1 Introduction

In the previous introduction Chapter 1, among other things, we discussed the major challenges of distributed networks. We then discussed the drawbacks of existing approaches and briefly presented our approaches. To address these major challenges of general distributed networks and to overcome the drawbacks of existing schemes and hence finish the majority of network flows more quickly, we present the design and analysis of a Quick cross-layer congestion Control Protocol (QCP). The QCP approach derives a simple and effective congestion control rate metric which routers calculate and at which sources share a link bandwidth to send data. This same rate metric is also used as a link weight metric to find a path for the content transfer flow. Unlike TCP, this rate metric can quickly obtain a very high link utilization and a low queue size and hence results in smaller AFCT. It can also achieve fairness (equal fair and proportional fair) among all flows quickly unlike XCP. QCP also uses an accurate derivation of the number of active flows and hence doesn't suffer from such estimation errors of RCP. QCP was called Fast Congestion control Protocol (FCP) [46] in our previous work as it is the fastest such protocol to converge to fairness and to full link utilization. QCP can be easily implemented using (extending) the OpenFlow [43, 44, 45] architecture.

QCP can emulate processor sharing (PS) [47] by dividing available link capacity fairly among flows. The PS scheme may not allocate a link capacity, unused by flows which are bottlenecked at other links, to other (local) flows which need the capacity. This is because in a multi-bottleneck network scenario, a traditional PS scheme does not have a good way to find out whether or not an active flow is bottlenecked locally or at another link. To deal with this drawback, we design an *Efficient Sharing* (ES) scheme by extending the QCP approach. The ES algorithm achieves this efficiency by treating each flow bottlenecked at other links (not locally) as a *fraction* of a flow.

Previous results [8] have shown how RCP outperforms XCP [17] and TCP. Our previous work at [39] presents numerical results showing that our cross-layer QCP scheme outperforms other schemes which use inverse of link capacity and delay as link weights and TCP as a congestion control protocol. The numerical

results in this chapter will focus on the congestion control aspect of QCP. More detailed results including path selection aspect of our cross-layer methodology are discussed in Chapters 3 and 4. The simulation results we present in this chapter show how QCP outperforms both XCP and RCP.

The main contributions of this chapter are as follows.

- We show that the performance of the Rate Control Protocol (RCP) degrades with network congestion.
- We propose a Quick congestion Control Protocol (QCP), which is a novel cross-layer congestion control scheme, that overcomes the weaknesses of existing congestion control schemes to achieve reduced flow completion time and high link utilization.
- We implement QCP in the NS2 simulator and present results which show how QCP outperforms RCP and XCP.
- We introduce a new resource sharing scheme called Efficient Sharing (ES) and generalized ES (GES) to achieve network level max/min fairness. ES can be more efficient than the traditional Processor Sharing (PS) in utilizing unused network resources without the need of multiple queues and complex schemes for flows. We show that QCP is an ES and GES protocol.
- We show how QCP can be implemented using (extending) the emerging OpenFlow [45] architecture which is currently being deployed in the backbone of major enterprise networks such as Google.

The rest of the chapter is organized as follows. The QCP algorithm is explained in section 2.2. In section 2.3 we present the derivation of the QCP rate. Sections 2.4 and 2.5 show how QCP can achieve processor sharing and even more efficient sharing (ES) than the traditional processor sharing (PS) [48]. The QCP packet header format is presented in section 2.6. We extend QCP to be a weighted fair share allocation in section 2.7. The stability of QCP is shown in section 2.8. We show how QCP can be gradually implemented in section 2.9. A description of how QCP can co-exist with TCP and TCP like algorithms is presented in section 2.10. After validating the performance of QCP using simulation in section 2.11, we discuss more related work in section 2.12. Finally, we give a brief summary in section 2.13.

2.2 QCP Algorithm

The QCP algorithm at the end-hosts and at the routers can be described as follows:

- A source sends each byte j with its desired rate $\hat{R}_j$ carried in the corresponding packet of the byte.

- Each router in the network calculates $R(t)$ using equation 2.2 or the simplified QCP version equation 2.12 every control interval d, $0 < d \leq RTT_{max}$. Here RTT_{max} is the maximum RTT of the flows which can be known or estimated offline.

- Each router in the path of a packet associated with byte j checks if $R(t) < \hat{R}_j$ in which case it overwrites $\hat{R}_j$ and forwards it unchanged otherwise.

- The destination then copies the $\hat{R}_j$ in the data packet to the ACK packet.

- The source sets its current window size $w'_j = \hat{R}_j RTT_j$ upon receipt of the ACK packet where RTT_j is the RTT of the flow of byte j.

- Each router updates its $R(t)$ value every control interval d.

- The rate $R(t)$ value is used a link weight by each router to obtain the path for the flows. The path computation can be done in a distributed fashion. In this case each router exchanges this rate value as its link state. Once the routers construct a topology map with link weights, they can then run a max/min shortest path algorithm such as the one at [39] to find the best path for the packets in the network. The routers can do this at some user defined regular intervals. Using the same rate value as a congestion control metric at the transport layer and as a link weight metric at the network layer makes QCP a cross-layer protocol.

2.3 QCP Rate

To define and derive the QCP rate metric, we first give notation in table 2.3.

Parameters	Description
C	Link capacity in bytes per sec
d	Duration of control interval in sec
$q(t)$	Queue size from the current interval in bytes
$R(t)$	Per flow rate allocation of the current interval in bytes per sec
N	Number of flows in the current interval
L	Total number of bytes which arrive to the router during a control interval of length d
m_i	Number of packets of flow i which arrive to the router
ϵ_i	The packet size of flow i in bytes
$\hat{L}$	Total number of packets which arrive to the router
R_i	Sending rate of flow i during the current round in bytes per sec
α, β	Stability parameters

Table 2.1: QCP Notations

The per flow fair QCP rate allocation at a bottleneck router is derived as follows. The intuition behind QCP rate is the assertion that the total number L of bytes sent to a router (link) during a control interval d shouldn't exceed the bandwidth-delay product minus the queue size at the router during the interval. Denote w_j to be the windows size of (number of bytes sent by) a flow of byte j. Byte j is carried by its packet which arrives at the router. Define the *per byte cwnd* to be the number of bytes to be sent in the next round trip time (RTT) for each of the w_j bytes sent by a source of byte j during the current RTT. If a source is sending at the rate R_j bytes/sec and plans to send at the fair share rate $R(t)$ bytes/sec in the next RTT, then the *per byte cwnd* is $R(t)/R_j$ bytes. The objective is to find the fair rate $R(t) = w'_j/RTT'_j$ using the current rate share $R_j = w_j/RTT_j$ of a flow associated with its byte j which arrives at the router.

The total number of bytes sent to a router from all sources in the next interval is then the sum of the *per byte cwnd* of all flows. With the notations defined in table 2.3, if $R_j = w_j/RTT_j$ denotes the rate associated with the jth of the L bytes which arrive to the router,

$$\sum_{j=1}^{L} \frac{R(t)}{R_j} = \sum_{k=1}^{\hat{L}} \frac{\epsilon_k R(t)}{R_k} = \sum_{i=1}^{N} \frac{m_i \epsilon_i R(t)}{R_i} = \alpha C d - \beta q(t) \tag{2.1}$$

This implies that

$$R(t) = \frac{\alpha C d - \beta q(t)}{\sum_{j=1}^{L}(1/R_j)} = \frac{\alpha C - \beta \frac{q(t)}{d}}{\frac{1}{d}\sum_{j=1}^{L}\frac{1}{R_j}}. \tag{2.2}$$

2.4 QCP Can Achieve Processor Sharing (PS)

In this section we discuss how QCP achieves PS. The inter-byte time σ_j is defined as the time between two consecutive bytes for a flow associated with byte j. It is given by $\sigma_j = \frac{1}{R_j}$. Now suppose a router has seen L bytes within the control time interval d. If n_i of these bytes carrying σ_i (in their corresponding packet) from source i are received by the router during the control interval d, then taking the denominator of equation 2.2 we have

$$\frac{1}{d}\sum_{j}^{L}\frac{1}{R_j} = \frac{1}{d}\sum_{i=1}^{N} n_i \frac{1}{R_i} = \sum_{i=1}^{N}\frac{n_i}{d}\frac{RTT_i}{w_i} \tag{2.3}$$

where N is the number of active flows and w_i is the congestion window size (cwnd) of flow i which is the number of bytes source i sends during its round trip time (RTT_i). The variable n_i is the total number of flow i bytes which arrive at the router during the control interval d.

In the case where all bytes sent from a source i at the rate of $R_i = w_i/RTT_i$ arrive to the next hop router (switch) at the same rate (as all the bytes of a flow to a router can be spaced at an equal interval of σ_i on

average) we have that

$$\frac{n_i}{d} = \frac{w_i}{RTT_i}. \tag{2.4}$$

This implies that $\frac{1}{d}\sum_j^L \frac{1}{R_j} = \sum_{i=1}^N (\frac{n_i}{d})/(\frac{w_i}{RTT_i}) = N$ which means that QCP can achieve PS.

In the next section we will discuss scenarios where $\frac{n_i}{d} \neq \frac{w_i}{RTT_i}$ and where QCP can perform better than the traditional processor sharing (PS) in a scheme we define as *efficient sharing (ES)*.

2.5 Efficient Sharing (ES)

Before we define *Efficient Sharing (ES)*, we will first describe the following notations. The resource to be shared has a capacity of $X\ units/sec$. Different sources use a sequence (chain) of different resources one after the other. Some sources are currently requesting a total of M units of the current resource of interest per interval τ. If there are N such sources and if source k requests n_k units then $M = \sum_k^N n_k$. A source associated with unit j has a bottleneck resource share rate denoted by $\Re_j\ units/sec$. So the source associated with unit j is sending requests to the current resource at the rate of $\Re_j$. The current rate allocation at the current resource of interest is denoted with $\Re(t-d)$. To define ES, set

$$\check{\Re}_j = max\left(\Re_j, \Re(t-d)\right) \tag{2.5}$$

where max is a maximum function.

Definition 1 *The Efficient Share allocation $\Re(t)$ for each source at the current resource of interest for the next interval is defined as*

$$\Re(t) = \frac{X}{\frac{1}{\tau}\sum_j^M \frac{1}{\check{\Re}_j}}. \tag{2.6}$$

In the case of QCP, $X = \alpha C - \beta\frac{q(t)}{d}$. We next show how ES outperforms the traditional PS and GPS by allocating capacity unused by some flows bottlenecked elsewhere to other flows which need the capacity. We also discuss how QCP is an ES protocol.

2.5.1 ES vs PS and GPS

A processor sharing (PS) which is also called a uniform processor sharing allocates a resource capacity of $X\ units/sec$ into N users equally. Its generalization is called Generalized Processor Sharing (GPS) [47] and was first proposed in [49] as weighted fair queueing (WFQ). GPS shares the resource X among the N users

according to their weights ϕ_j. A source (user) i gets the share $\Re_i(t)$ given by

$$\Re_i(t) = \frac{\phi_i}{\sum_k^N \phi_k} X. \tag{2.7}$$

The case where all the ϕ_j are the same is a PS scheme. The weight values of ϕ_j are picked by the GPS scheduler. However GPS does not give any specific approach to obtain the *weights* in such a way that a resource m which can not be used by a user which is bottlenecked at another resource n is allocated to another user which can use the resource m. On the other hand, ES uses the weights given in the denominator of equation 2.6 to implicitly assign unused resource to all flows which can use it.

Even though ES obtains the same rate to all flows as shown in 2.6, the flows which do not need the assigned rate implicitly release the resource to other flows which require it by not using the capacity of the resource beyond what they can use (bottlenecked elsewhere). For example if $X = 100\ units/sec$, $R_1 = 10\ units/sec, n_1 = 10\, units$, $R_2 = 50\ units/sec, n_2 = 50\, units$ and $R_3 = 70\ units/sec, n_3 = 70\, units$ for a control interval of $\tau = 1\ sec$, and $R(t-d) = 40\, units/sec$ then since $max(10, 40) = 40$ from equation 2.5,

$$\Re(t) = \frac{100}{\frac{10}{40} + \frac{50}{50} + \frac{70}{70}} = \frac{100}{2.25} = 44.44\, units/sec.$$

Here, all three users are assigned the same rate $\Re(t) = 44.44$. However user 2 implicitly releases the resources it can not use by sending only at 10 $units/sec$. A PS mechanism would result in a share of $100/3 = 33.3$ and would not assign the resource unused by user 1 to the other users (sources). The GPS on the other hand does not provide a mechanism to obtain proper weight values at a multi-bottleneck level to allow efficient use of resources. Hence ES can also be viewed as a special case GPS where weights are automatically and adaptively calculated at a distributed network level in such a way that what some sources (users) *can not use* is *equally allocated* to other users in a work conserving manner (utilizing available resource if there is a demand for it).

A resource some users cannot use can also be allocated to other users which can use it proportionally based on some weights ϕ_i as the case of GPS. We call this generalization of ES a generalized efficient sharing (GES). The same argument as equation 2.1 can be used to find a new rate $\Re_j(t)$ with weight ϕ_j associated with unit j as follows.

$$\sum_k^M \frac{\Re_k(t)}{\Re_k} = \sum_k^M \frac{\phi_k \Re(t)}{\Re_k} = X\tau. \tag{2.8}$$

This implies that

$$\Re(t) = \frac{X}{\frac{1}{\tau}\sum_{k}^{M}\frac{\phi_k}{\Re_k}}. \tag{2.9}$$

Then the GES rate $\Re_j(t) = \phi_j \Re(t)$.

In addition to achieving PS, QCP can also handle ES scenarios which a traditional PS scheme cannot handle. This enables QCP to achieve more efficient sharing (ES) than PS. We will next use two QCP cases to describe this ES.

2.5.2 Single Bottleneck Scenario

There can be a scenario where $\frac{n_i}{d} < \frac{w_i}{RTT_i}$. This happens for instance when a new bottleneck link is formed in the flow path before the location of the previous bottleneck link which allocated $R_i = \frac{w_i}{RTT_i}$ to flow i. The new bottleneck link then drops or delays packets of flow i resulting in smaller rate $\frac{n_i}{d}$ arriving to the previous bottleneck link. In this case, QCP in the previous bottleneck link counts flow i as less than one flow (*fractional flow*) which is equal to $\frac{n_i}{d}/R_i$. On the other hand, the traditional PS counts each of such flows as one flow. In this case, the PS approach at the previous bottleneck link divides the capacity by more than the *actual* number of flows. This results in PS allocating less rates to some flows which need more. Dividing the capacity by the *exact fractional number* of flows, QCP however gives the capacity unused by some flows to flows which can use it without causing buffer overflow or resource underutilization. To do this, QCP doesn't require any special queues or complicated operations as the allocation is done using QCP rate equation. On the other hand, the scenario where $\frac{n_i}{d} > \frac{w_i}{RTT_i}$ may occur, for instance when bursts of packets of a flow arrive to a link. In this case, QCP temporarily counts such flows as $\frac{n_i}{d}/R_i$ which is more than one flow. Hence, QCP assigns less rate to flows to absorb the bursts of packets.

Another important result from equations 2.4 and 2.3, is that unlike RCP [8] and XCP [17] the estimation of the control interval, d, in QCP *doesn't need the exact flow RTTs*. The value of d can be set to some reasonable value between maximum and minimum RTT values. It can be user-defined and obtained from reasonable offline experiments. The smaller the value of d, the more recent bottleneck rate values the packets carry back to their sources. QCP is less sensitive of the choice of d. This is because if the choice of d results in $\frac{n_i}{d} \neq \frac{w_i}{RTT_i}$, the ES nature of QCP temporarily counts the flow as a fractional flow resulting in an accurate rate calculation as discussed above. QCP can also use flow RTTs to obtain d like XCP and RCP.

2.5.3 Multi-Bottleneck Network

In a multi-bottleneck network where different flows are bottlenecked at different links, some flows may not be able to utilize their equal share allocation at a link which is a bottleneck to other flows. If the bottleneck link allocation of flow i is R_i and if its current equal rate share at its non-bottleneck link is $R(t-d) > R_i$, then flow i can waste its non-bottleneck link capacity which can otherwise be used by other flows bottlenecked at that link. This can result in QCP not achieving ES.

To deal with this scenario, QCP uses

$$\check{R}_j = max(R_j, R(t-d)) \tag{2.10}$$

instead of the R_j in the denominator of equation 2.2, where R_j is the source rate carried by a packet associated with byte j of a flow and $R(t-d)$ is the rate allocation of flows at the link for the current interval.

QCP uses expression 2.10 only if the flow associated with byte j is in its second RTT sending packets at its bottleneck link rate. QCP can check this by comparing R_j against the initial QCP rate R_{init} of flows which can be known before hand as the ratio of initial *cwnd* and some average flow RTT. In this case if $R_j \leq R_{init}$, QCP doesn't use the expression 2.10 as the flow may be just starting. QCP packet header can also carry a single bit to indicate the start of a flow. If possible, SYN packet can also be used to indicate the start of the flow. OpenFlow switches (routers) can also detect the first packet of a flow if the packet does not belong to any of the flow table entries [45].

Here is some explanation of why the approach in expression 2.10 can achieve ES (Efficient Sharing). If $R_j < R(t-d)$, then a flow which owns byte j should be treated as a partial (fractional) flow by the router which allocated $R(t-d)$ to the flows (including the flow of byte j). This enables QCP to assign the unused resource to other flows bottlenecked at that router.

On the other hand, if $R_j > R(t-d)$, QCP achieves ES by treating the flow of byte j as at least one flow as it can cause temporary queue spikes (being late to learn its new allocation). This occurs for instance because the allocation R_j was much older than $R(t-d)$ as the flow has an RTT too long (longer than the control interval) to know about its latest rate allocation.

If we approximate R_j used in equation 2.2 with $R(t-d)$ even if $R_j > R(t-d)$, we get

$$\begin{aligned} N_a &= \frac{1}{d}\sum_{j}^{L}\frac{1}{R_j} \approx \frac{1}{d}\sum_{j}^{L}\frac{1}{R(t-d)} \\ &\approx \frac{1}{R(t-d)}\frac{L}{d} = \frac{y(t)}{R(t-d)} \end{aligned} \tag{2.11}$$

where $y(t) = \frac{L}{d}$ is the total input traffic rate in bytes during the control interval d at the router, and $R(t-d)$ and R_j are rates per flow.

When QCP uses equation 2.11, it can overestimate the actual number N_a of flows when $\frac{1}{R_j} < \frac{1}{R(t-d)}$. This overestimation of N_a can result in a lower rate allocation to all flows which in turn can result in link underutilization. This is specially true with a misbehaving flow. If QCP uses equation 2.11, the misbehaving flow can continue to increase its rate at the expense of the other flows as the router continues to count the flow as more than one flow. Such behavior is not fair to the other flows which quickly obey the QCP rate rule.

In a simplified version of QCP, we also use equation 2.11 in the denominator of equation 2.2 as an estimation of the actual number of flows. The resulting simplified QCP rate is then given by

$$R(t) = \frac{\alpha C d - \beta q(t)}{d N_a} \tag{2.12}$$

The derivation in equation 2.11 shows that the main strength of this simplified version of QCP lies on its use of the *fractional flow* concept where flows can be counted as partial flows unlike the case of PS. Hence, the simple expression given by equation 2.11 is an estimator of ES. This implementation allows QCP packet header to be even smaller (about 8 bytes) as shown in section 2.6. In the simulation experiments of this chapter we used the exact QCP rate given by equation 2.2.

2.6 QCP Packet Header Format

The QCP header can be placed as shim layer between the TCP and IP headers. QCP can have two packet header implementation schemes. The first one which is shown in figure 2.1 has a 12 byte header.

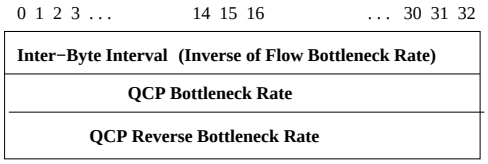

Figure 2.1: QCP header with 12 bytes

The first field is the *Inter-Byte Interval* length $\sigma_j = 1/R_j$, where R_j is the current sending rate attached to a packet associated with byte j of the corresponding flow. The routers in the path of byte j (its associated packet) use this field to obtain the QCP rate given by equation 2.2. The second field is the *QCP Bottleneck Rate* $\hat{R}_j$ which is the rate initialized to be the desired rate by source. The bottleneck router in the path of the packet associated with byte j can then overwrite the value. This rate is the minimum of all the rates

in the path of the packet associated with byte j. The third field is *QCP Reverse Bottleneck Rate* which is the same QCP bottleneck rate which the receiver copies to its outgoing packets (ACK packets for example). The simulation results for QCP used in this chapter use this implementation scheme of the QCP header.

The second implementation scheme of the QCP header shown in figure 2.2 is without the σ_j field. This implementation can reduce the QCP packet header to 8 bytes.

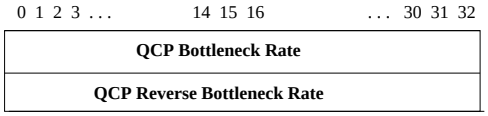

Figure 2.2: QCP header with 8 bytes

In this implementation scheme each source sets the value of the *QCP bottleneck rate* ($\hat{R}_j$) to its desired rate. Each router in the path of the packet associated with byte j calculates the rate using equation 2.12. If this rate is smaller than the $\hat{R}_j$ in the packet header, then the router replaces the $\hat{R}_j$ in the packet header with what it obtains using equation 2.12. The receiver then copies the value of the *QCP bottleneck rate* which routers may have changed, into the ACK (returning) packets. The source which receives the ACK packets then adjusts its *cwnd* to the product of the rate it gets from the ACK packets and its RTT.

2.7 Weighted QCP

The QCP rate given by equation 2.2 can be extended to be a weighted share metric. Such a metric allows different flows to get different shares based on their weights without causing router buffer overflow or link under-utilization. If packet j of a flow carries the weight information ω_j of its flow, then using a derivation similar to the one used in equations 2.8 and 2.9, we have

$$\sum_{j=1}^{L} \frac{\omega_j R(t)}{R_j} = \alpha C d - \beta q(t). \tag{2.13}$$

This implies that

$$R(t) = \frac{\alpha C d - \beta q(t)}{\sum_{j=1}^{L} (\omega_j / R_j)}. \tag{2.14}$$

A flow of packet j can then set its rate $R_j = \omega_j R(t) = \omega_j \hat{R}_j$ where $R(t)$ is the bottleneck link rate.

Different policies can be set for different classes of flows. For instance, if a flow which just received the rate of $\hat{R}_k$ from its ACK packet k wants to achieve a target rate of $R_j^{\tilde{T}}$ for the next round, it sets its weight ω_j as $\omega_j = \frac{R_j^{\tilde{T}}}{\hat{R}_k}$.

Different levels of priority can be used by adding a few more bits in the QCP header or using the current IP header fields (ECN bits). The source can also send ω_j/R_j in the QCP header. Each source i can then set its congestion window as $w_i = \omega_i \hat{R}_i RTT_i$ packets where $\hat{R}_i$ is obtained from the ACK packets.

2.8 Stability Analysis

In this section we present stability analysis of QCP using control theory.

2.8.1 Lyapunov Stability

The rate allocation by QCP queue at a bottleneck router is done every control interval d. This allocation is received by each source sharing the bottleneck link after a round trip time (of each of the sources). This new rate allocation changes the congestion window w_j of each source j. So the aggregate feedback sent per unit time is the sum of the derivatives of the congestion windows. This feedback is similar with the XCP feedback and hence we have

$$\sum_j \frac{dw_j}{dt} = C - \Lambda(t-d) - \frac{q(t-d)}{d} \tag{2.15}$$

where $\Lambda(t-d)$ and $q(t-d)$ are the total arrival rate and queue size in the previous control interval and C is the link capacity.

Adding the control parameters α and β for stability, Equation 2.15 becomes

$$\sum_j \frac{dw_j}{dt} = \alpha(C - \Lambda(t-d)) - \beta\frac{q(t-d)}{d}. \tag{2.16}$$

As shown in [50] and [51], the QCP feedback mechanism is given by the delay differential equations

$$\begin{aligned} \Lambda'(t) &= \frac{\alpha}{d}(C - \Lambda(t-d)) - \frac{\beta}{d^2}q(t-d) \\ q'(t) &= \begin{cases} \Lambda(t) - C, & q(t) > 0 \\ max\{\Lambda(t) - C, 0\}, & q(t) = 0. \end{cases} \end{aligned} \tag{2.17}$$

As the QCP feedback mechanism can be written in Equation 2.17, appropriate Lyapunov functions can be used to find stable values of the control parameters α and β. For instance the work [51] shows that $\beta/d^2 = \alpha/d$ gives stability. This for instance implies that if $\alpha = 1.0$, $\beta = d$. Previous work [50] also shows a wide range of stable values for protocols whose feedback mechanism can be written in the form of

Equation 2.17. Our detailed simulation results also show that $\alpha = 1 = \beta$ gives stable values for QCP. We are also working on using the Lyapunov functions in [51] to find even wider stable regions for α and β.

2.9 Gradual Deployment of QCP

In the implementation of QCP, routers (router-like boxes) read the QCP packet headers, calculate rate and modify the packet headers of flows. QCP can be easily implemented by extending the OpenFlow [43, 45] which enables clean slate schemes to be implemented in big networks such as the Google backbone network [52, 53]. In this section we will discuss how QCP can be implemented using (by extending) the OpenFlow architecture.

There are different ways QCP can be implemented using the OpenFlow switch and protocol specification [45]. For the rest of this section we will use the terms switch and router interchangeably. As specified in [45], each flow entry of an OpenFlow switch contains a set of *instructions* that are executed when an arriving packet matches the entry. One of the instruction types is *Meter* which directs the packet to a specified meter. Each meter has one or more *meter bands*. Each band specifies the *rate* at which the band applies and the way packets should be processed. Packets are processed by a single meter band based on the *current measured meter rate*. The meter applies the meter band with the highest *configured rate* that is lower than the current measured rate. If the *current measured rate* is lower than any *specified meter band rate*, no meter band is applied.

In the case of QCP, the configured rate can be obtained by taking the *QCP bottleneck rate* ($\hat{R}_j$) from the QCP packet header. The *measured meter rate* associated with a specific link can be replaced with the QCP rate in equation 2.19 obtained as discussed in section 2.9.1. The OpenFlow switch can then invoke the *Apply-Actions* instruction to apply the *Set-Field* action which overwrites the QCP bottleneck rate in the packet header.

2.9.1 QCP Rate Using OpenFlow

The flow table of an OpenFlow switch can maintain a per flow packet counter. By polling the packet counter every control interval d the number n_i of packets of each flow i during the interval d (for each of the N flows sharing a given link) can be obtained. The flow table also maintains the numbers L_r of received and L_s served bytes for each flow from which the queue size $q(t)$ of a link can be obtained. The $q(t)$ can also be obtained by reading the OpenSwitch queue length. By reading n_i at every control interval d, the current sending rate r_i of flow i is given by $r_i = n_i/d$. If the QCP rate of the link obtained from the previous control

interval is $R(t-d)$, then setting

$$\check{r}_i = max\,(r_i, R(t-d)), \tag{2.18}$$

the QCP rate can then be calculated as

$$R(t) = \frac{\alpha C - \beta \frac{q(t)}{d}}{\sum_{i=1}^{N} \frac{r_i}{\check{r}_i}}. \tag{2.19}$$

These counters can be obtained and the rate can be calculated using the OpenFlow switches (decentralized) with off-ASIC or on-ASIC CPU [54, 55, 56]. It can also be calculated by the OpenFlow controllers (centralized) and sent to the OpenFlow switches. The OpenFlow switches then apply the *Set-Field* action using the instruction discussed above. The *Set-Field* action can be applied to the first packet of each flow allowing each source to jump start its sending rate as what Quick start TCP [57] aims to do. The *Set-Field* action can also be applied to some randomly selected or to all packets of a flow. This implementation scheme can be done using the 8 byte QCP header scheme as discussed in section 2.6.

2.9.2 Using Stateless OpenFlow

Another QCP implementation scheme using the OpenFlow concept can use the 12 bytes or 8 bytes QCP header scheme discussed in section 2.6. In this implementation scheme, the OpenFlow switches do not even need to read the bytes counts from the flow tables. In this case, the source rate R_j in the QCP packet header field which carries $\sigma_j = \frac{1}{R_j}$ with equations 2.2 and 2.10 is used instead of the r_i in equations 2.18 and 2.19. This implementation scheme does not require OpenFlow switches and controllers to maintain per flow states (such as counters), which is the main OpenFlow scalability issue [58]. The per link rate which can be calculated using equation 2.2 (with end-host assistance) or equation 2.12 (with no end-host assistance), is all the switches and controllers need to achieve QCP and other OpenFlow objectives. For instance, OpenFlow rate limiting and max-min routing can be done as briefly discussed in sections 2.9.4 and 2.9.3, using this rate.

2.9.3 Using OpenFlow Edge Switches

The QCP rate, given by equation 2.19 or 2.2, can also be used as a link weight metric in a max-min routing algorithm. The max-min routing algorithm finds the minimum rate of each path and takes the path with the highest such rate. The OpenFlow controller can run the max-min algorithm and provide the edge OpenFlow switches with such max-min *path* and the corresponding *rate*. Parallelism and other approaches are being used to scale the OpenFlow controller[59, 60, 61]. The max-min can also be done by the routers

(OpenFlow switches) in a distributed manner by exchanging the rate as a link metric. The edge switches can then use the *Set-Field* action to replace the rate at the QCP packet headers. The QCP packet headers then do not have to be changed until they reach another edge switch (router) which has rate of its link. Using this approach saves the core switches (routers) more packet processing time. This approach can be easily performed using emerging network virtualization architectures such as [62] with intelligent edge open vSwitches and controller cluster.

2.9.4 QCP Rate Limiting

Rate limiting of a flow can also be done using the QCP rate. If the *measured rate* R_M of a flow at a switch is higher than the QCP rate $R(t)$ obtained using equation 2.19, packets of the flow can be dropped (sent to low priority queue) with probability $(R_M - R(t))/R_M$. Otherwise, the packets are served with no drops. Different flows sharing a link can also get different rate allocations to support Quality of Service (QoS). By associating a weight ω_i to flow i the QCP rate can be obtained as

$$R(t) = \frac{\alpha C - \beta \frac{q(t)}{d}}{\sum_{i=1}^{N} \frac{\omega_i r_i}{\check{r}_i}}. \tag{2.20}$$

Flow k can then get a share of $R_k = \omega_k R(t)$. In this case, if $R_M > R_k$, packets of flow k are dropped (sent to low priority queue) with probability $(R_M - R_k)/R_M$. Otherwise, packets of flow k are served with no drops.

2.10 QCP with TCP Flows

To allow QCP to be implemented with other protocols such as TCP for incremental deployment, the QCP router can be modified as follows. The QCP router creates separate fair queues for TCP and QCP traffic. The router serves packets from the TCP and QCP queues based on weights, for instance, using round robin. The weights ω_T and ω_Q of the TCP and QCP queues can be calculated the same way as the weights of TCP and XCP queues are calculated in [17]. To force its flows to be fair to TCP, QCP also uses the remaining capacity $\omega_Q C$ instead of the entire link capacity C in the calculation of the QCP rate using equations 2.2 and 2.20.

Table 2.2: Baseline parameters for experiments on estimation of N

Parameter	Default value
Link capacity	20 Mbps
Link propagation delay	50 ms
Number of flows	10
File size	4 MB

2.11 Simulation Analysis

In previous studies [8], RCP was shown to outperform TCP and XCP. In this section, we evaluate the performance of QCP comparing it with RCP and XCP using NS2 [63] which is a widely used network simulator.

To validate the performance of QCP, we have implemented the QCP source as a sub-class of TCP-Reno and QCP queue as a subclass of DropTail Queue in NS2.

Similar to previous work on RCP, we first use a simple topology which contains sources and destinations connected by one single bottleneck link. Unless specified we use a router buffer size of 1 bandwidth-delay-product (BDP).

In the first set of experiments, we show how RCP and QCP make estimation on the number of flows. We generate a fixed number of big size flows which all start at the same time. The baseline parameters are summarized in Table 2.2.

Figure 2.3 plots the estimation of number of flows versus time for QCP and RCP. We use the same value of $\alpha = 1 = \beta$ for QCP in all experiments while RCP uses different values of α and β in different experiments. The estimation of N from QCP virtually matches the real value. In contrast, depending on the choice of parameters, the estimation of N from RCP either needs much longer time to converge or even never converges. This in turn results in flows taking longer to finish than necessary. Two important messages conveyed here are: (1) QCP gives a more accurate and reliable estimation of the number of flows than RCP; (2) the performance of RCP is sensitive to the setting of parameters and there is no specific rule on how to set these parameters. In the rest of the chapter we use $\alpha = 0.1$ and $\beta = 1$ for RCP experiments.

To compare the average flow completion time (AFCT) of QCP against RCP we have also considered a different numbers of flows with a fixed file size. As can be seen from Figure 2.4 the AFCT of QCP is smaller than that of RCP.

RCP performs badly when the number of flows grows as shown in figure 2.4. This is because RCP either under or over estimates the number of flows into which the bandwidth is divided.

Most of the experiments used to validate RCP in [8] were obtained using a non-congestion scenario with an average link load of around 90%. However such a simulation scenario doesn't properly evaluate the

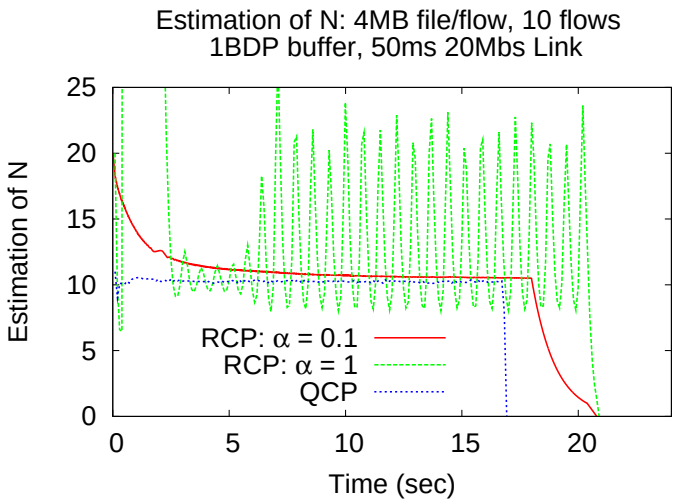

Figure 2.3: Estimation of N versus time with $\alpha = 0.1$, $\alpha = 1$ for RCP

Protocol	Number of finished flows (in 26.061 sec)
RCP	17280
QCP	66212

Table 2.3: QCP versus RCP under high load scenario: Poisson(8333.3 flows/sec), Pareto(1.2,30 pkts)

performance of RCP. In fact as in a *Naive QCP* approach, where we set the initial *cwnd* of every flow equal to the file size of the flow for the cases, where the link on average is not fully utilized (similar to many RCP experiments in the [8]), the network doesn't get congested on average as shown in figure 2.5. In this scenario a congestion control protocol is not even strictly needed as all flows can send all the packets they have in one round and retransmit some of their lost or delayed packets to get very small AFCT. As can be seen from the plot, even Naive QCP outperforms RCP.

However, under a congestion scenario, the performance of RCP is worse when compared with QCP as shown in the next experimental results. In these experiments Poisson flow arrivals where the file sizes are Pareto distributed are used as is also the case in [8, 17, 15] to emulate Internet and data-center traffic [5]. As shown in table 2.3, within a simulation time of 26.061 seconds only 17280 RCP flows finished due to the increasingly high file completion time (FCT) as shown in figure 2.6. On the other hand as can be seen from table 2.3, 66212 QCP flows finished during the same time.

The y-axis of figure 2.6 shows how the average completion time of flows that complete within each progressing 2 second interval grows with simulation time in a loaded link scenario. As the simulation time progresses, RCP results in higher file completion time of the flows that finish. This in turn results in less flows finishing in RCP than in QCP as shown in table 2.3.

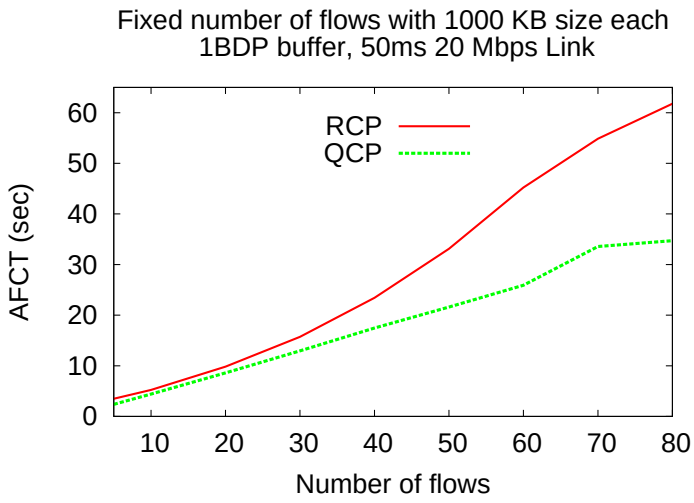

Figure 2.4: AFCT versus number of flows

QCP_M	XCP_M	QCP_E	XCP_E	QCP_{Avg}	XCP_{Avg}
0.4001	0.8971	3.8176	3.5130	1.5962	1.8127

Table 2.4: AFCT (seconds): QCP versus XCP

We also compared the FCT of the 17280 RCP flows (all RCP flows) which finished against the first 17280 QCP flows which finished. As shown in figure 2.7 the FCT of QCP flows is much smaller than that of RCP. This small FCT helped more QCP flows finish in a shorter time as shown in table 2.3.

We next give comparison of QCP against XCP which is another major clean slate congestion control protocol. As discussed in section 2.12.2, XCP is not fair to small size flows (called mice) which are the majority of Internet flows. This is because the link bandwidth is dominated by a few large size flows (called elephants).

Table 2.4 shows that the AFCT of 20 flows about one-third of which are elephant flows and the remaining 13 flows are mice flows. The file size of each of the elephants is 1MB and that of each mice is 50KB. The single bottleneck link bandwidth is 20 Mbps with a propagation delay of 50ms. All the elephant flows start at the same time and the mice flows start about 1 second after the elephant flows start.

As shown in table 2.4, the AFCT of XCP mice flows (XCP_M) is more than twice the AFCT of QCP mice flows (QCP_M). This shows how unfair XCP is to short flows when compared to QCP. While achieving this fair allocation to small file size (mice) flows, QCP does not compromise the overall average flow completion time (QCP_{Avg}) when compared with XCP. QCP is also fair to the big file size (elephant) flows (QCP_E).

Figures 2.8 and 2.9 present the CDF of QCP versus XCP for a link capacity of 2.4Gbps with a propagation delay of 50ms. In figure 2.8 flows arrive following a Poisson distribution with mean 6000 flows/sec and file

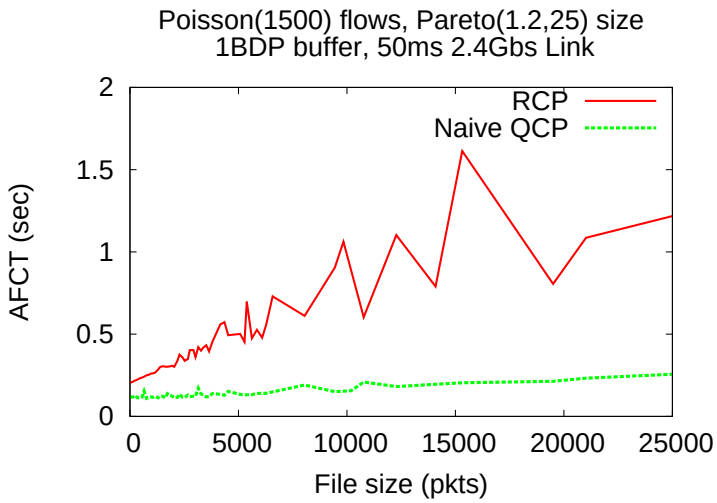

Figure 2.5: AFCT of *Naive QCP* vs RCP: Poisson(1500 flows/sec), Pareto(1.2, 25pkts)

sizes are Pareto distributed with mean 50 packets (packet size = 1000 Bytes) and shape parameter of 1.6. In figure 2.9 flow arrival is Poisson with mean alternating between 5400 flows/sec for 3 seconds and 30000 flows/sec for 2 seconds. This simulates a link where the average load fluctuates between high load and low load. In figure 2.9 file sizes are also Pareto distributed but with mean 30 packets and shape parameter of 1.2.

As can be seen from both figures 2.8 and 2.9 the file completion time of of the majority of QCP flows is smaller than that of XCP flows. As file sizes are Pareto distributed to emulate the Internet flows, most of the flows in this simulation setup are small size flows (mice).

We have also compared the performance of QCP against RCP for a two bottleneck network topology as shown in figure 2.10.

As shown in figure 2.11, QCP gives lower FCT for the two groups of flows crossing two different bottleneck links as shown in the topology of figure 2.10.

We have also simulated QCP against RCP and XCP using flow inter-arrival times and flow size traces taken from [5] and [64]. In figure 2.12 we use a bottleneck link capacity of 0.4Gbps and flow inter-arrival times uniformly distributed between 10 and 100 micro seconds taken from [5] to evaluate QCP against other protocols under a high load (congestion) scenario. Like the previous experiments, QCP flows finish quicker resulting in more QCP flows finishing. Within 8.6 simulation time, 48511 QCP flows and 37762 RCP flows completed. Figure 2.12 shows the FCT CDF of all RCP flows that finished against the corresponding QCP flows that finished.

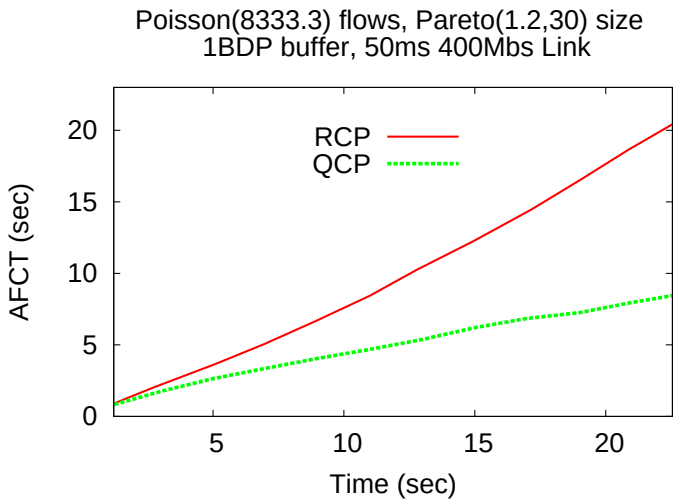

Figure 2.6: FCT of flows versus simulation time (with 2 sec aggregation): Poisson(8333.3 flows/sec), Pareto(1.2, 30 pkts)

2.12 Related Work

In this section we discuss previous work on congestion control protocols. We start with TCP and its variants and then analyze the major clean-slate congestion control protocols.

2.12.1 TCP and its Variants

The performance limitations of TCP over high bandwidth delay product networks has been reported in [14]. They showed that a random packet loss can result in a significant throughput degradation. The same paper also shows that TCP is unfair towards flows with higher round trip delays. TCP is also not fair for short-lived flows as shown in [13] as the bottleneck bandwidth is dominated by long-lived flows whose window size has grown so large. As has been extensibly reported in the literature [65], TCP is also not suitable for wireless networks. The main reason is that TCP assumes that all packet losses are due to network congestion while in the case of wireless networks it can be due to some wireless link errors which may soon correct themselves. TCP also either under-utilizes or over-utilizes the network bandwidth resulting in a download time much longer than necessary as discussed in [8].

The current modifications to TCP inherit the main problems of the original TCP and have not properly addressed the main challenges. The Datagram Congestion Control Protocol (DCCP) [66] which is primarily designed to replace the User Datagram Protocol (UDP) whose unreliable nature can cause congestion collapse is for instance based on the TCP algorithm. There are also many other variants of and modifications to TCP [67, 68, 16, 69]. Nonetheless these modifications of TCP inherit the basic limitations of TCP of not

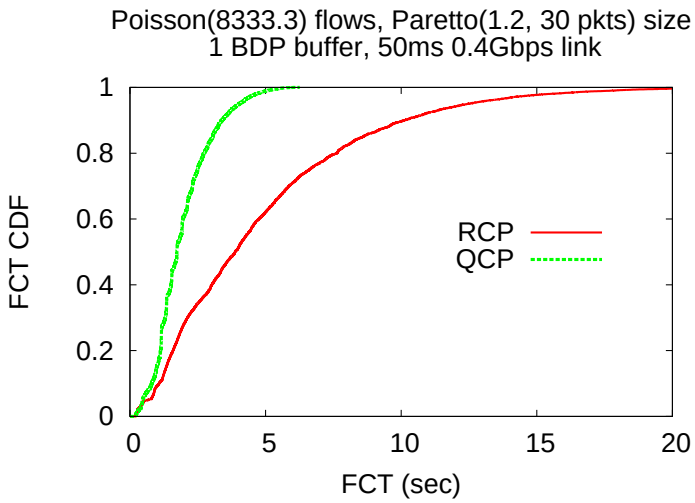

Figure 2.7: FCT CDF of finished RCP flows vs QCP flows: Poisson(8333.3 flows/sec), Pareto (1.2, 30pkts)

quickly knowing the bottleneck link share of flows in spite of some improvements over the original TCP. They mainly rely on packet loss and packet delay as congestion signals. Hence, they take longer than necessary to fully utilize the link capacity and to achieve fairness among flows. This in turn results in higher average flow completion (download) time (AFCT).

2.12.2 Major Clean Slate Protocols

In this section we discuss how QCP differs from RCP and XCP which are the two other major clean slate congestion control protocols.

On Performance of RCP

Using the notations, d as a moving average of the RTTs measured across all packets arriving at a link, C as the link capacity, $q(t)$ as the instantaneous queue size at a router of the link, $N(t)$ as the router's estimate of the number of ongoing flows at time t and with α, β as parameters chosen for stability and performance, the rate update equation of the rate control protocol (RCP) [8] is given by

$$R(t) = R(t-d) + \frac{(\alpha(C - y(t)) - \beta\frac{q(t)}{d})}{N(t)} \tag{2.21}$$

where $R(t-d)$ and $y(t)$ are the the updated rate and the measured input traffic rate in the previous update interval respectively.

In RCP and in the rate control protocol with acceleration control (RFC-AC) [70], the number of ongoing

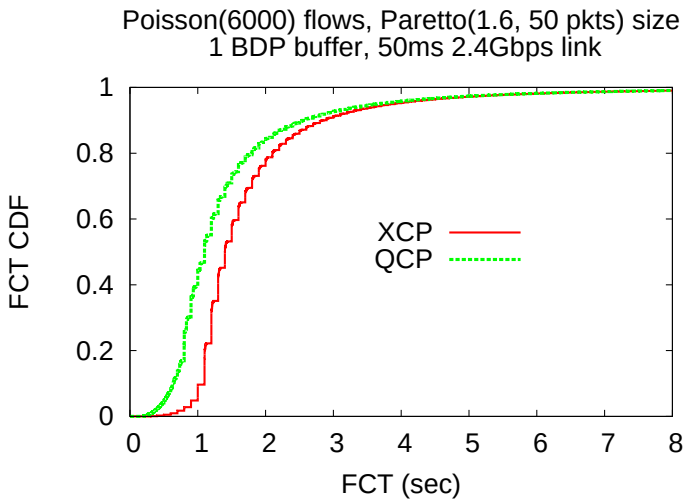

Figure 2.8: FCT of XCP flows vs QCP flows: Poisson(6000 flows/sec)

flows, $N(t)$ is estimated as

$$N(t) = \frac{C}{R(t-d)}. \tag{2.22}$$

But this is a heuristic estimate and is where the major limitation of RCP lies.

RCP either over-estimates or under-estimates the allocated rate $R(t)$. When the initial value of $R(t-d)$ from which $N(t)$ is obtained is too small, then $N(t)$ is too large . This in turn results in the router unnecessarily dividing the capacity into too many flows resulting in link under-utilization. Let's consider an initial rate of $R(t-d) = C/200$ whose corresponding $N(t) = 200$. If the link receives only 40 flows/sec for an RTT of 0.1 sec, we have an actual number of 4 flows. If the router allocates each of these flows only $C/200$ bytes/sec, then the total arrival rate for the next round becomes $C/50$ bytes/sec which is 1/50 of the available link capacity. In this case RCP significantly under-utilizes the link capacity.

On the other hand if the initial value of $R(t-d)$ is too large, then $N(t)$ becomes too small. As a result the router divides the capacity into fewer number of flows and hence over-estimates the rate allocation. This causes link over-utilization, more queuing delays and packet losses. In fact, the simulation setup of RCP uses a big buffer capacity (to try to deal with this).

For example, let the initial sending rate $R(t-d) = C/4$. Then the corresponding $N(t) = 4$. If the flow arrival rate is 200 flows/sec for an RTT of 0.1 sec, the actual number of flows is 20. The router then tells each of these 20 flows to send at the rate of $R(t-d) = C/4$. If they all send at this rate, then the total arrival rate $\Lambda = 20C/4 = 5C$. Hence the link receives 5 times more packets than it can handle.

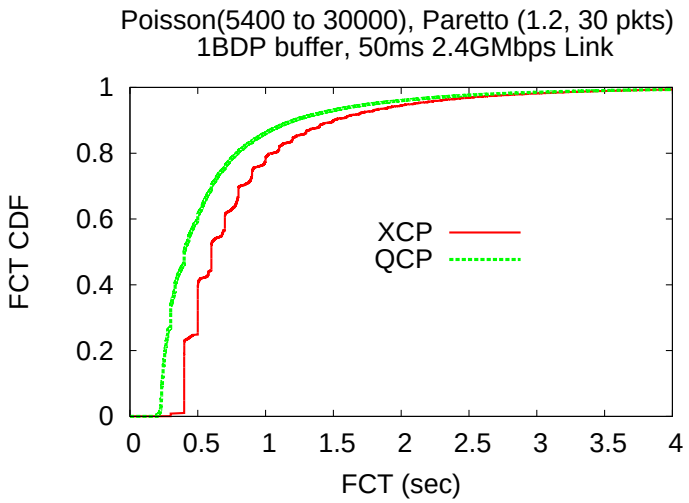

Figure 2.9: FCT of XCP flows vs QCP flows: Poisson(5400 to 30000 flows/sec)

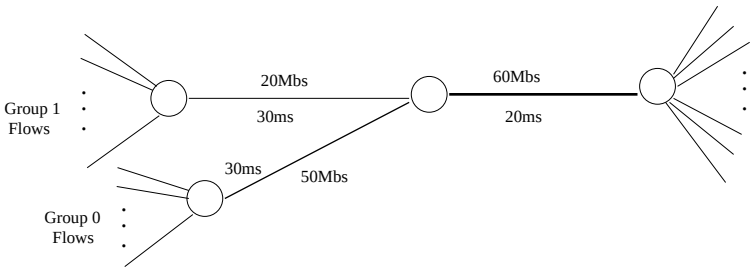

Figure 2.10: Two bottleneck network topology

On Performance of XCP

The fact that XCP is not fair to short flows (flows with small data to send) makes its average flow completion (download) time (AFCT) much higher than TCP as shown in [8]. For example, let us consider three short lived flows (mice) which just started with a congestion window size of 1 packet and need to send 50 packets each and one long lived flow (elephant) which needs to send 500 packets and already reached a congestion window size of 60 packets. Without loss of generality let's assume that they all have the same round trip time (RTT). If the spare link capacity is 20 packets per RTT, then XCP shares it equally among all four flows allowing each flow to increase its congestion window by 4 packets per RTT. This implies that the window size of the three short lived (mice) flows is now set to 5 packets per RTT. Hence, it takes $50/5 = 10$ rounds (RTT) to download each of the short lived flows and hence a longer AFCT for most of the flows. But QCP can reduce this FCT of majority of the flows by dividing the entire link capacity (say $60+20 = 80$ packets/RTT) equally among all four flows. This implies that each flow sets (resets) its window size to $80/4 = 20$ packets per RTT. This implies that each of the short lived flows (the majority) will have a file

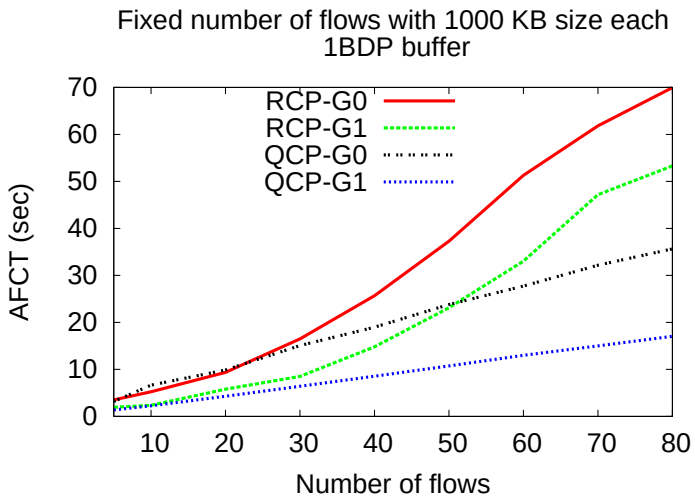

Figure 2.11: FCT of Group 0 (G0) and Group 1 (G1) RCP and QCP flows

download time of about 2.5 rounds (RTT).

2.13 Summary

In this chapter we first discuss how existing congestion control schemes can result in higher average flow completion time (AFCT) than necessary. We then presented the design of a Quick congestion Control Protocol (QCP) to more quickly finish flows. QCP uses a *fair rate metric* to determine *the rate at which flows send data* and to select the path for the flows. We have shown how QCP can quickly achieve network level max/min fairness using a more efficient sharing (ES) scheme (hence lower AFCT) than the traditional processor sharing (PS). QCP can assign different rates to different flows using source specified weights. We have discussed the stability and TCP-friendliness of QCP. We have described how QCP can be implemented in the OpenFlow networking architecture which is currently being deployed in major enterprise networks.

Results from implementation of QCP in the NS2 simulator show that QCP can outperform RCP and XCP which are the two major clean slate congestion control protocols in terms of AFCT by upto 30%.

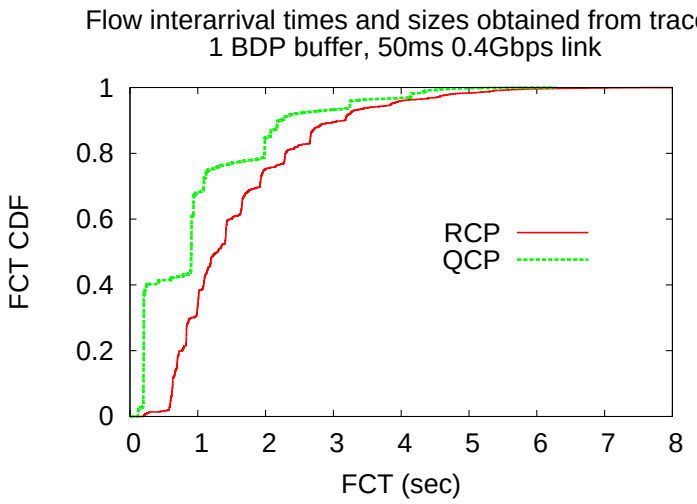

Figure 2.12: FCT CDF of RCP flows vs QCP flows: Flow inter-arrivals and sizes taken from traces

Chapter 3

SCDA: Cross-Layer SLA-aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval

3.1 Introduction

In the previous Chapter 2 we presented a cross-layer QCP protocol for general networks. In this chapter we present the design of a cross-layer SLA-aware Cloud Datacenter Architecture (SCDA) for efficient content storage and retrieval. SCDA among other things addresses the challenges cloud datacenter networks discussed in Chapter 1. The design of SCDA has two main features. The first feature enables SCDA to use multiple name node servers (NNS) using a light weight front-end server (FES) which forwards requests to the name nodes (NNS). This approach solves the weakness of current state-of-the-art cloud-computing architectures (file systems) [28, 29]. In such systems only a single NNS, which can potentially be a bottleneck resource and single point of failure, is used.

The second main feature of SCDA is its ability to avoid congestion and select the less loaded servers using a cross-layer (transport and network layers) concept unlike current well known schemes [28, 29, 25, 26] which rely on TCP and random server selection. SCDA also uses resource monitors (RM) and resource allocators (RA) to do fine grained resource allocation and load balancing. The roles of these SCDA components can be extended to constantly monitor the performance of the cloud against malicious attacks or failures. All the aggregated and monitored traffic metrics can be offloaded to an external server for off-line diagnosis, analysis and data mining of the distributed system.

The data center (cloud) resource *allocation and enforcement* mechanism of SCDA using RMs and RAs is *stateless* and does not need modifications to routers/switches or the TCP/IP stack. The scheme can *detect violation in service level agreements (SLA)* and can help cloud (data center) administrators (admins) to automatically add more resources to resolve detected SLA violations.

The SCDA resource (bandwidth) allocation mechanism is *max-min fair* in that any link bandwidth unused by some flows (bottlenecked elsewhere) can be used by flows which need it. This is a very useful quality any resource allocation mechanism needs to achieve. We also show how SCDA can do more *power aware server selection* as there is heterogeneity in power consumptions by different servers. This heterogeneity can

be due to server's location in a rack, due to server age and specifications or due to other compute intensive or background tasks the servers perform. The RM and RA of SCDA are *software components* and can be consolidated into a few powerful servers close to each other to minimize communication overheads and latencies.

The rest of this chapter is organized in such a way that we first present the network and content models used in the design of SCDA in section 3.2. In section 3.3 we discuss the SCDA nodes and software components. Section 3.4 discusses simple formulas to obtain the rate metric at which clients share resources (link bandwidth, CPU, storage). The steps used in the SCDA algorithm are presented in section 3.5. In the subsequent sections 3.6, 3.7 and 3.8 we discuss each of these steps. These steps are the rate allocation mechanism at a global and different levels of the data center tree, the server selection mechanism using the allocated rate metrics and ways to serve both outside client and internal cloud requests respectively. In section 3.9 we give a brief description of how SCDA can be applied to different cloud network topologies. In section 3.10 we present experimental results comparing the performance of SCDA against existing schemes which use random server selection and TCP (RandTCP). A related work discussion is given in section **??**. We finally give summary of the chapter in section 3.12.

3.2 Network and Content Model

In this section we present description of the network and content models for which we design and analyze SCDA.

3.2.1 Network Model

Under SCDA, the network consists of client nodes connected to cloud data-center servers via links. The clients are connected to the cloud via dedicated tunnels as part of the service level agreement (SLA) or over the Internet. This is usually done using protocols such as the OSPFv3 as a Provider Edge to Customer Edge (PE-CE) Routing Protocol [71]. The cloud data-center servers are connected with each other via high speed links typically using a hierarchical topology similar to the one shown in figure 3.1. SCDA can be easily extended to work with other data-center network topologies such as [25, 72] as briefly described in section 3.9.

3.2.2 Content Model

Contents stored in cloud data centers can be classified into *active* and *passive*. A *passive content* is content which is not frequently read or written to, after its initial storage in the cloud. An *active content* on the other hand is a content which is frequently accessed due to read or write actions. The read and write frequencies to distinguish passive content from active content in our design are user defined parameters. Active contents can further be classified into *high write and high read* (HWHR), *low write and high read* (LWHR) and *high write and low read* (HWLR). Following this classification, the passive contents can be considered as *low write and low read* (LWLR). Considering an email application for instance, sent emails and attachments can be considered passive for the sender. Chatting (both text and video/audio) can be considered active content. A file which is edited by collaborative users can be considered an active content. Database tables which are constantly updated can be considered active contents. Some hot news can be considered an active content.

As shown in [73] for HDFS logs in one of Yahoo!s enterprise Hadoop clusters, about 60% of content was not accessed at all in a 20 day window. Hence SCDA takes content diversity into account when selecting storage or replica servers for each request to store or retrieve content. SCDA uses different server selection strategies for the active and passive contents.

3.3 SCDA Components

The architecture of SCDA [74] is presented in figure 3.1. As shown in the figure, the SCDA architecture assumes a tree structure of the data center networks for cloud computing as is the case with most data center networks today. Our SCDA scheme also works with other cloud and data center network topologies. The solid lines in figure 3.1 show the physical cable connections. The arrows show logical control flow communications between SCDA components. Like existing popular large scale distributed file systems [28, 29], SCDA consists of a network of block servers (BS). Unlike GFS and HDFS, SCDA uses a light weight front end server (FES) and more than one name node server (NNS). This enables SCDA to solve the potential problems of GFS and SCDA in being bottlenecked at the single NNS. SCDA also achieves its efficient resource allocation and load balancing schemes and energy efficiency using rate monitors and rate allocators. We next discuss the nodes and the resource monitors and allocators of SCDA.

3.3.1 Nodes

The nodes in SCDA consist of the front end server (FES) which receives external requests to and from the local cloud and forwards them to the respective name node server (NNS). Each NNS keeps metadata

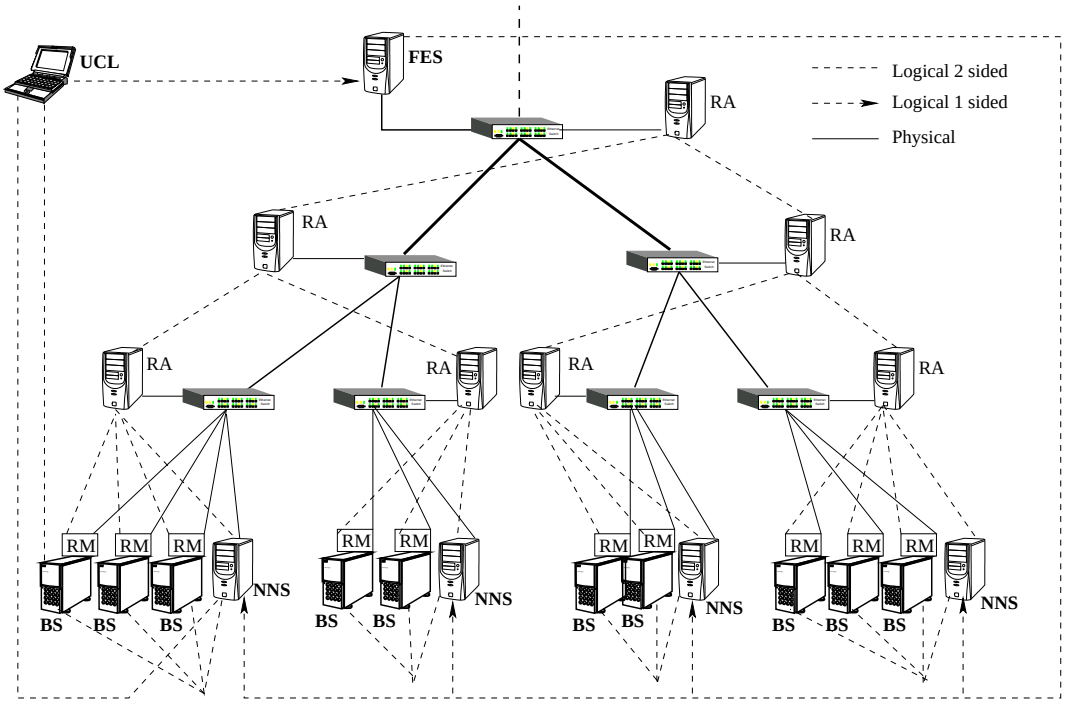

Figure 3.1: SCDA Architecture

information, for example, which block of data is stored in which block server (BS). Each BS stores data blocks assigned to it by the NNS. To help balance load among all NNS, the FES may also be assisted by the NNS to forward requests to other NNS. The UCL node is a user client which requests cloud services. The functionality of FES can be moved to the UCLs or to the NNS. FES agents associated with the UCL clients can forward the client requests to the corresponding NNS. When an FES agent is associated with the NNS, a UCL can connect to any of the NNSs. If the hashing function maps the UCL request to the receiving NNS, the NNS serves the request. Otherwise the NNS hashes the request and forwards it to the corresponding NNS. Multiple FES with different IP address for different regions can also be employed. The DNS then chooses the nearest matching FES.

3.3.2 Resource Monitors and Allocators

The resource monitor (RM) of SCDA is a software component responsible for monitoring and sending resource load information from the BS to the resource allocators (RA). The RAs on the other hand gather resource load information from each BS via the RMs and other information from the switches and calculate SCDA rate allocation metrics at each level of the tree.

3.4 SCDA Rate Metric

To define the rate metric, we first present the following notations.

For each SCDA parameter $X \in \{R, C, Q, \hat{N}, N, n^j, R^j\}$, we use the notation

$$X_{d,u} = \begin{cases} X_d & \text{if } X \text{ is a downlink parameter,} \\ X_u & \text{if } X \text{ is a uplink parameter.} \end{cases} \tag{3.1}$$

We next give short descriptions of the SCDA parameters.

Variables	Description
$C_{d,u}$	Link capacity in bits/sec
τ	Control interval in sec
$Q_{d,u}(t)$	Link queue size from the current interval (round) in bits
$R_{d,u}(t)$	Link rate allocation of the current interval in bits/sec
$N_{d,u}(t)$	Number of flows in the link during the current round
$\hat{N}_{d,u}(t)$	Effective number of flows in the link for the current round
$R^j_{d,u}(t)$	Rate of flow j for the current round in bits/sec
$S_{d,u}(t)$	Sum of flow bottleneck rates in current round in bits/sec
$\Lambda_{d,u}(t)$	Total current arrival rate to the link in bits/sec
$L_{d,u}(t)$	Total number of bits at a link in the current interval
$\wp^j_{d,u}$	Priority weight of flow (stream or chunk) j
$M^j_{d,u}$	Minimum rate requirement of content flow j
α, β	Stability parameters

Table 3.1: SCDA Parameters

Given the above SCDA parameters, each RA and RM calculate the rates $R_d(t), R_u(t)$ of the down (d) and up (u) links associated with their local switches as follows:

The down-link (d) and up-link (u) rates

$$R_{d,u}(t) = \frac{\alpha C_{d,u} - \frac{\beta Q_{d,u}(t-\tau)}{d}}{\hat{N}_{d,u}(t-\tau)} \tag{3.2}$$

where

$$\hat{N}_{d,u}(t-\tau) = \frac{S_{d,u}(t)}{R_{d,u}(t-\tau)}, \tag{3.3}$$

$$S_{d,u}(t) = \sum_{j}^{N_{d,u}(t-\tau)} R^j_{d,u}(t) \tag{3.4}$$

and

$R^j_{d,u}(t) = min\left(R^j_{send,other}(t), R_{e2e}, R^j_{recv,other}(t)\right)$. Here, the R_{e2e} is the end-to-end link rate of flow j obtained using max/min algorithm discussed in section 3.6.1 below. The $R^j_{send,other}(t)$ and $R^j_{recv,other}(t)$

are the flow rates at the sender and receiver sides of the tree due to other bottleneck resources (CPU computation, disk storage). The CPU of the server which sends or receives flow j may be too busy with internal computations to serve external write or read requests at the e2e link rate, R_{e2e}. Or the server may not have enough disk space. The application generating flow j may also not have enough data to send or cannot send at the e2e link rate.

As shown in Figure 3.1, each RA and RM get the values of $Q_d(t-\tau)$ and $Q_u(t-\tau)$ from the local switch (router) with which they are connected (associated). This doesn't need any change to the switches as all switches maintain the queue length in each of their interfaces. Each RM computes the effective number of up-link and down-link flows using equation 3.3. Each RM reports the values of $S_d(t)$ and $S_u(t)$ to its parent RA. Each RA adds these values from each of its children to find its $S_d(t)$ and $S_u(t)$ values. Each RA also sends its accumulated sum $S_d(t)$ and $S_u(t)$ for both the down-link and up-links to its parent RA. This continues until the highest level RA. After the first time RM sends its $S_d(t)$ and $S_u(t)$ values, it can send the difference Δ_d and Δ_u values to its parents for all other rounds (if there is a change in the rate values). This is to minimize the overhead by sending the difference which is a smaller number than the the sum of the rates. Each RA can also do the same by sending the difference instead of the actual effective number of flows to its parent RA.

Each RM and RA perform the computation of equation 3.2 periodically every control interval τ. This control interval for the RM can be estimated as the average of the round trip times (RTT) of the flows of its BS or it can be a user defined parameter. For instance the maximum RTT can be used. Each RA at level h computes its $R_{d,u}(t)$ after it gathers the $S_d(t)$ and $S_u(t)$ information from all its children or after a certain timeout value T_o expires.

Equation 3.3 enables SCDA to be a *max-min fair* protocol where resources (link bandwidth) unused by flows bottlenecked at other resources (links) can be utilized by flows which need it. For instance, if $R_u^j(t)$ is a bottleneck rate of flow j which is not bottlenecked at a link which allocated $R_u(t-\tau)$, then this link counts flow j as $\frac{R_u^j(t)}{R_u(t-\tau)}$ which is less than 1 flow.

The simplified SCDA rate metric can also be given by

$$R_{d,u}(t) = \frac{(\alpha C_{d,u} - \beta \frac{Q_{d,u}}{d}) R_{d,u}(t-\tau)}{\Lambda_{d,u}(t)} \tag{3.5}$$

where $\Lambda_{d,u}(t) = L_{d,u}/\tau$ is total packet arrival rate to the router during the control interval τ. In this simplified version of SCDA each RA and RM can also get the values of $L_{d,u}(t)$ from the corresponding switch or router. Hence, for this simplified version of SCDA, the RMs and RAs do not need to report the rate sum values $S_d(t)$ and $S_u(t)$ of flows to their parent nodes (RA).

3.4.1 Prioritized Rate Allocation for a Desired QoS level

SCDA can also achieve a desired quality of service (QoS) value by allocating different rates to different flows. This is done by using the priority $\wp^j_{d,u}$ weight of flow j in equation 3.4 as shown by equation 3.6.

$$S_{d,u}(t) = \sum_{j}^{N_{d,u}(t-\tau)} \wp^j_{d,u} R^j_{d,u}(t) \tag{3.6}$$

The source of each flow specifies the priority weight values using the RM to achieve a desired rate value. If the source j gets the bottleneck rate $R^j_{d,u}(t)$ discussed above (section 3.4) and if it wants to set its rate in the next round $t+\tau$ to $R^j_{d,u}(t+\tau)$, it sets it priority as

$$\wp^j_{d,u} = \frac{R^j_{d,u}(t+\tau)}{R^j_{d,u}(t)}.$$

This way a source can achieve the desired rate of its flow j by increasing or decreasing the corresponding $\wp^j_{d,u}$. This approach can adaptively and implicitly implement many scheduling policies in a distributed manner. For instance something like a shortest file (job) first (SJF) and early deadline first (EDF) scheduling algorithms can be implemented by assigning higher target rate $R^j_{d,u}(t+\tau)$ for short or early deadline flows resulting in higher priority weight $\wp^j_{d,u}$ for such flows.

Equation 3.6 is also very important for *detecting and ensuring service level agreement (SLA)*. A SLA violation is *detected* if the sum $S_{d,u}(t)$ of a link exceeds the link capacity $capacity = \alpha C_{d,u} - \beta \frac{Q_{d,u}}{d}$ in equations 3.2 and 3.5. The RM detects SLA violation if its $S_{d,u}(t)$ exceeds the *capacity* of the link it is associated with.

We denote RM level of the tree with level 0. The RAs which are direct parents of RMs are at level 1. The highest level RA is at level h_{max}. The value of h_{max} of figure 3.5 is 3. The RAs at level 1 detect SLA violation if the sum of $S_{d,u}(t)$ from their children RMs exceeds the *capacity* of the link they are associated with. The RAs at levels higher than 1 detect the SLA violation if the sum of $S_{d,u}(t)$ from their children RAs exceeds the *capacity* of the link they are associated with. Once the SLA violation is detected, the corresponding RM or RA automatically requests for more bandwidth allocation in its link or other alternative links. The SLA violation report by an RM or RA can also be handled by the NNS assigning a different BS for the requesting node. Such selected BS must have enough available bandwidth to support the new request. The data center can also maintain reserve, backup or recovery links to resolve SLA violations automatically. The weights of prioritized flows can then be adaptively adjusted by each distributed source at every RTT to achieve the desired rate of a specific flow.

3.4.2 SCDA with OpenFlow

SCDA can also implement the QoS Prioritization by using the functionalities of current OpenFlow switches [45]. Each OpenFlow switch maintains packet count Cnt_j for each flow j. So to implement SJF scheduling, the switch can approximate a small size flow to be the flow which has sent fewer packets. The OpenFlow switch then always serves the packets of the flow with smaller packet count. As the packets of the flows which already sent more packets are delayed, such flows reduce their sending rates due to delayed ACK packets. Each RM can also send the priorities of its flows to its RA. The RA can then inform the OpenFlow switch to schedule the packets of the flows according to the priorities.

3.4.3 QoS By Explicit Reservation

In the SCDA scheme, some sources can also reserve minimum rate $M^j_{d,u}$ required. In this case, for each requesting flow j, the total available link capacity to be shared by other flows is reduced by $M^j_{d,u}$. So if we have $N^{Res}_{d,u}$ such flows reserving capacity, the $C_{d,u}$ in equations 3.2 and 3.5 is replaced with $C_{d,u} - \sum_k^{N^{Res}_{d,u}} M^j_{d,u}$. Each RM first sums the $M^j_{d,u}$ values of its node. It then sends the sum to its RA. Each RA also sends these values to its parent RA. When the top level RA receives the sum of the reservations, all flows will have their desired bandwidth reserved in the data center (cloud). The remaining capacity can then be allocated among all flows sharing links.

3.5 SCDA Algorithm

The SCDA algorithm adaptively performs

- per flow resource (link, storage, processing) rate allocation at a global and h-level of the cloud data center tree,
- decision of how and where in the cloud to store data using the allocation information,
- decision of which (replica) server in the cloud to retrieve stored data from and how.

In the following sections we discuss each of these SCDA algorithms in detail.

3.6 Global and h-Level Rate Allocation

Each NNS needs to decide (a) which BS at level h to choose to store each block of data and (b) at what rate to send data from one BS to another BS or to/from an external agent. To do this, the NNS asks the

RA at level h, $0 \leq h \leq h_{max}$ of the tree as shown in figure 3.1. Hence each RA needs to maintain the best down-link and up-link rate values and the address of the BS or BSes with these rate values. For global allocation, the highest level values are needed. Here h_{max} is the maximum level value in the tree like cloud topology starting from the BS nodes. For such three tier topology, $h_{max} = 3$. In such topology, the block servers (BS) are at level 0.

Each NNS among other things also needs to decide at what rate to replicate data from one BS in one level of the cloud tree to another BS in another part of the cloud by asking each RM. Hence each RM also needs to keep the up-link and down-link bottleneck rate values upto each level of the tree. To achieve this, each RA forwards its rate values obtained using equation 3.2 to its children. Besides, each RA needs to forward to its children the minimum of its rate and the rates forwarded to it from its higher level parents. Finally, these rates of each level of the cloud tree are received by each RM.

The above best h-level rate values stored at each RA and RM are obtained using a max-min scheme as follows.

3.6.1 Obtaining the Rate values using Max/Min Algorithm

Here is how the rate metric at different levels of the network tree are obtained as also described in figure 3.2.

To get the metrics kept by the RAs:

- Each RM j at level $h = 0$ sets its downlink (d) and uplink (u) $\hat{R}_{d,u}^{hj}$ rate values to its the minimum of $R_{d,u}^{hj} = R_{d,u}(t)$ which is obtained using equation 3.2 or equation 3.5 and its R_{other}^{hj}. The rate value R_{other}^{hj} is a function of the CPU and disk loads. If either the available CPU speed or disk speed are too low, R_{other}^{hj} decreases accordingly. For instance R_{other}^{hj} can be measured from the previous control interval. It can as well be the weighted average of previous intervals. The CPU and disk usage can be profiled to get what CPU and/or usage can serve what link rate. This approach allows SCDA to be a *multi-resource allocation* mechanism. If link bandwidth is the only bottleneck resource, we set $\hat{R}_{d,u}^{hj} = R_{d,u}^{hj}$.

- The RMs also calculate $S_{d,u}^{hj} = S_{d,u}(t)$ using equation 3.6.

- Each RM sends its $\hat{R}_{d,u}^{hj}$ and $S_{d,u}^{hj}$ values to its parent RA which is associated with the switch the RM and RA are directly associated with (connected to).

- Each RA j at level h calculates its $S_{d,u}^{hj}$ by summing up the $S_{d,u}^{(h-1)j}$ of its children in the RA-RM tree as shown in figure 3.2. The RA then calculates its $R_{d,u}^{hj} = R_{d,u}(t)$ using equation 3.2.

- Each RA j at level h sets its $\hat{R}_{d,u}^{hj}$ to the minimum of its $R_{d,u}^{hj}$ and the highest $\hat{R}_{d,u}^{(h-1)j}$ obtained from its children.

- Each RA j at level h then stores its $\hat{R}_{d,u}^{hj}$ values and sends them to its parent RA along with the ID of the corresponding BS. The parent also does the same.

- By the time this process reaches the RA at level h_{max} which is the highest level RA associated with with a switch/router at the entry point to the cloud, each RA j at level h has the best h-level $\hat{R}_{d,u}^{hj}$ and the ID of the corresponding best BS. These values are useful for the NNS to decide *where to store (write) data*.

To get the metrics kept by the RMs:

- The highest level RA (at level $h = h_{max}$) sends its $R_{d,u}^{hj}$ values along with its level number down to its children RAs. Each (child) RA at level $h-1$ also forwards the minimum of its rate and each of its higher level rates along with the level numbers to its children. Finally each lowest level RA forwards these values to its children RM.

- At this point each RM knows the best h-level up-link and down-link rate values $\tilde{R}_{d,u}^{hj}$ along with the level numbers. These values are helpful for the NNS in deciding where *to read replicated data from* and to update the *rates of on-going flows* to and from the main cloud (data center) using the information in the RM.

3.7 Cloud Server Selection

After the rate values are obtained using a max-min algorithm discussed in the above section and stored at each RA and RM, SCDA selects a cloud server to store the data of a requesting client. The server is selected in such a way that transfer, retrieval and processing of the data is fast and efficient. To achieve this goal, SCDA treats active and passive contents discussed in section 3.2.2 differently. Passive data is with low write and low read (LWLR) frequency. *Interactive content* is where write and read operations are interleaved in less than a few seconds interval with high frequency (HWHR). We consider a maximum interactivity interval of 5 seconds to decided whether or not a content is interactive. A *semi-interactive content* is either with HWLR or LRHW. The client applications can specify the type of content or the RMs of the servers can learn the type of content from the server access frequencies (of writes and reads) by the content. We next show the server selection strategies for each type of content.

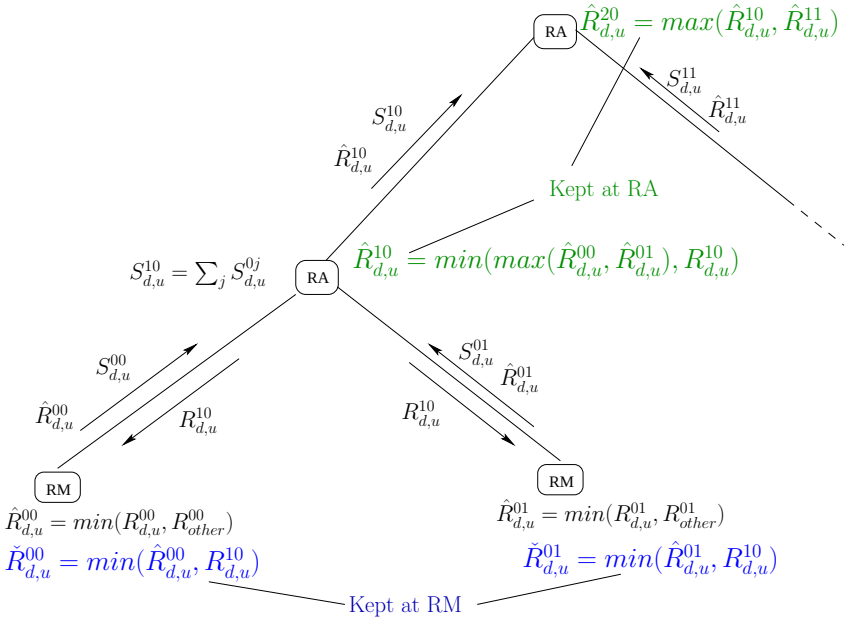

Figure 3.2: The SCDA Max/Min Scheme

3.7.1 Interactive Content

For interactive applications, the RA at level $h+1$ also keeps the highest of the $min(\hat{R}^{hj}_{d}, \hat{R}^{hj}_{u})$ of all its children RM or RA where min is a minimum function. Here, $\hat{R}^{hj}_{d}$ is the downlink rate and $\hat{R}^{hj}_{u}$ is the uplink rate of a link at level h (child node of the RA) as shown in figure 3.2. This is because for interactive applications, the rate at which the interaction is done is limited by either the uplink rate to or downlink rate from the selected server, whichever is smaller. As this process goes up the RA tree hierarchy, all RAs, including the highest level RA (at level $h = h_{max}$), keep $\hat{R}^{hj}_{min} = min(\hat{R}^{hj}_{d}, \hat{R}^{hj}_{u})$. SCDA then chooses a BS with highest $\hat{R}^{hj}_{min}$ to serve requests for interactive contents.

3.7.2 Semi-interactive Content

For semi-interactive applications where either the write or the read operations is very frequent, the server selection is done in two stages. In the first stage, the RA chooses the server at level h with the best downlink rate $\hat{R}^{hj}_{d}$. This server is the server to which content (data) writing by clients is the fastest. In the second stage, the server to which data is being written chooses another replication server with the best uplink rate $\hat{R}^{hj}_{u}$. This ensures that the content retrieval is fast. So for these kind of applications, writing is done to the server where data transfer and writing is the fastest. Content reading (retrieval) is done from the (replica) server which offers the fastest reading (upload) rate.

3.7.3 Passive Content

A content with low write and read frequency (passive content) is replicated at *dormant* servers. *Dormant* servers are in low-power, high-energy-saving inactive power modes as there are more idle periods of server utilization. By sending passive content to dormant servers, SCDA saves energy by reducing latencies associated with the power state transitions. So SCDA can save energy by scaling down some servers with passive content.

Server selection for passive content requests is also done in two stages. In the first content write stage, the server with the highest download rate $\hat{R}_d^{hj}$ is selected. This ensures that data is written fast. The server to which this data is written then selects a replication server which has an upload rate $\hat{R}_u^{hj}$ greater than the scale down threshold rate R_{scale}. The value of R_{scale} is user specified depending on how aggressive the scale down needs to be. For highly aggressive scale down R_{scale} is small. This scale down value can also be set adaptively.

As long as there are passive contents, interactive and semi-interactive contents do not use servers whose upload rates $\hat{R}_u^{hj}$ are greater than R_{scale}. For these applications the RMs of servers to which data is written, select other servers with $\hat{R}_u^{hj} < R_{scale}$ for content replication. This leaves the least loaded servers (servers with very high $\hat{R}_u^{hj}$) for the passive data. This essentially keeps the dormant servers dormant resulting in effective scale down of servers.

Passive content which is initially written to the active servers can be totally moved to the dormant servers after the active servers learn the low frequency of the content. The RM of the active servers to which data is initially written can obtain the frequency (popularity) of contents by counting the number of accesses.

3.7.4 More Power Efficient Server Selection

In this section we will discuss how to handle heterogeneity in servers energy consumption. This heterogeneity can be due to location of a server in a rack or room, specifications and age of the server hardware and other (processing) tasks the server is doing [75]. So SCDA takes such diverse energy consumption by each server into account while selecting server for each requesting client application. To do this, SCDA relies on measurements of each server's energy consumption. This measurement can be done by (heat or temperature) sensors in the servers. Denoting the heat measurement at time t with $H(t)$, the power consumption during a control interval τ is given by $P(t) = \frac{H(t)}{\tau}$.

Each RA j of SCDA at level h can then select a server with the highest rate to power ratio by replacing $\hat{R}_{d,u}^{hj}$ in section 3.6.1 with $\frac{\hat{R}_{d,u}^{hj}}{P(t)}$. Other functions of power and rate can also be used to perform server selection. The value of $P(t)$ can be obtained as a running average or with more weight to the latest power

consumption measurement.

We next show the steps involved in serving write and read requests by an external requesting client and by the internal cloud (data center) servers.

3.8 Serving Requests

In this section we discuss how requests for cloud data center resources are served. The requests can be external to write to and read data from the cloud servers. The request can also be internal to replicate or move data from one cloud server to another server in the same cloud. We next discuss how SCDA serves such requests.

3.8.1 Serving External Write Request

To serve an external request of a user client (UCL) to use cloud resources, SCDA performs the following steps which are also presented in figure 3.3.

1. The UCL sends its ID (IP Address) along with the request to write into a cloud (data center) server.
2. The FES hashes the UCL ID and forwards it to the corresponding NNS. The matching NNS can for instance be the server with the ID equal to *hash(UCL_ID) mod* N_{NNS} where N_{NNS} is the number of NNS in the cloud data center and *mod* is the modulo operation.
3. The NNS asks the RA at a level where it wants to select a cloud server from. If the NNS wants to select a server at a specific rack, it asks the RA at level 1 of the corresponding rack for the best server in that rack. This best server is the server to which sending data is the fastest among those in the rack. If the NNS wants to select the best server in the data center, it asks the RA at level $h_{max} = 3$ for the best server (3 tier).
4. The RA selects a block server (BS) with the best rate.
5. The NNS then forwards the UCL ID to the selected BS.
6. The selected BS asks its RM for the download rate all the way from the highest level router (RA) in the data center (cloud).
7. The RM responds with rate which it obtained from the highest level RA via intermediate RAs.

8. The BS sets its receive window size (*rcvw*) to the product of the downlink rate it obtained from its RM and the RTT of the flow. The initial value of the RTT can be updated with more packet arrivals. The receiving cloud server can obtain the RTT from the time stamp values in the headers of the packets it receives from the sender.

9. The selected BS then contacts the requesting peer (UCL) to start the connection which the UCL uses to write data to the BS. While doing so, the the BS sends its receive window size (rcvw) in the packet header.

10. The UCL asks its RM for the upload rate.

11. The RM responds with the upload rate which is the minimum rate upto the highest level RA router (switch).

12. The UCL then sets its congestion window (cwnd) to the product of the upload rate and its RTT. As the UCL also receives the rcvw from the destination BS, it sets its sending window size to the minimum of the *cwnd* and *rcvw*. If the UCL has no RM (no dedicated tunnel), then setting the *rcvw* can ensure that the UCL does not send more than what the datacenter can handle.

13. The UCL then starts writing its data to the selected server.

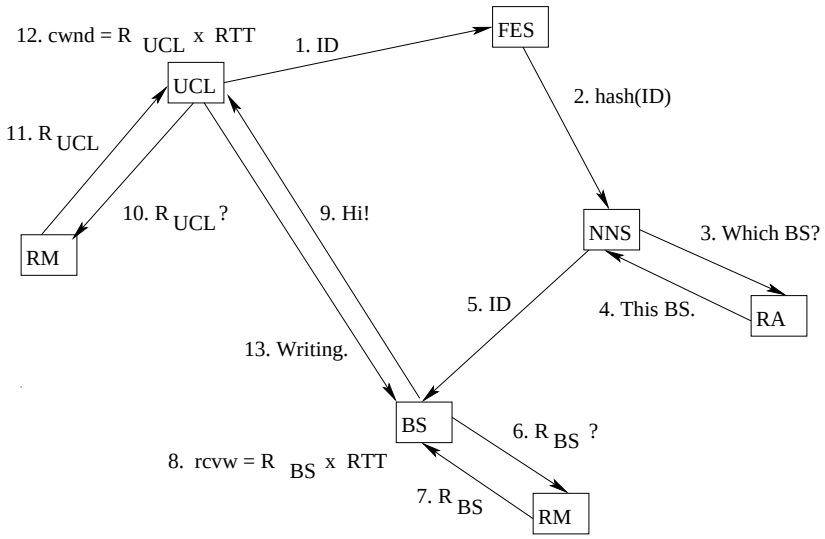

Figure 3.3: Serving External Write Request

3.8.2 Serving Internal Write Request

Once a UCL writes (uploads) data content to the cloud BS which offers the best upload rate, the BS decides to replicate or move the content to another BS which can offer the best upload rate with minimum energy consumption. To do this SCDA follows the following steps which are also described in figure 3.4.

1. First the BS (e.g. BS11) which wants to replicate the content contacts the NNS of the content by sending hash of the content ID.
2. The NNS selects a block server (BS23) based on the content selection algorithm discussed in section 3.7 (server which offers high upload rate) to ensure that future client read requests are fast.
3. The rest of the steps are similar with the steps in serving external write request discussed in section 3.8.1 with BS11 instead of the UCL and BS23 as the BS in figure 3.3.

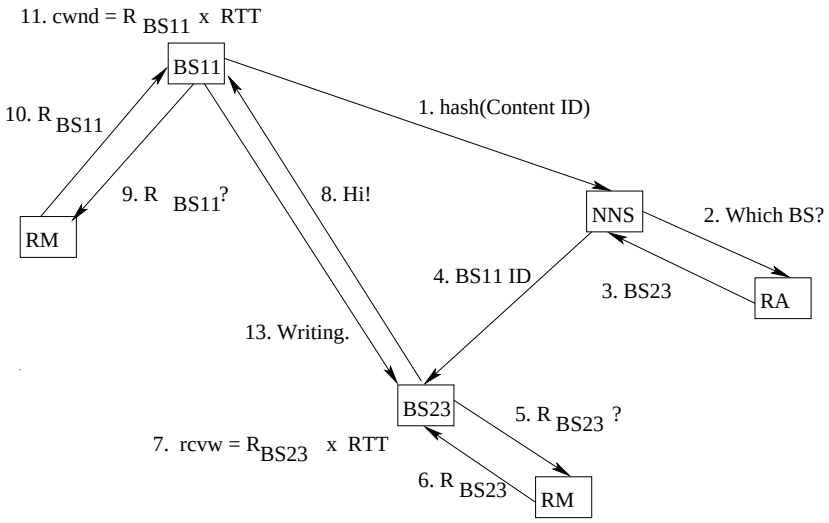

Figure 3.4: Serving Internal Write Request

3.8.3 Serving External Read Request

Once the UCL writes data to the selected cloud BS and after this BS replicates the content, the UCL request to read data is served using the following steps which are also presented in figure 3.5. The server selection is based on the server selection mechanism discussed in section 3.7.

1. The UCL requests for a certain content to read from the cloud by sending its ID.
2. The FES hashes this UCL ID and forwards it the responsible (corresponding) NNS which has the metadata of the requested content.

3. The NNS can either maintain the best BS for each of the contents whose metadata it keeps. It can also request (poll) the RMs of the BSs which have the content for their upload rates. It then chooses the best BS based on the server selection mechanism discussed in section 3.7 (server with the content which has high upload rate). The NNS forwards the UCL ID to the selected BS.

4. The selected BS asks its RM for the upload rate.

5. The RM provides its BS with the upload rate.

6. The BS sets its cwnd to the product of this rate and its RTT. If the UCL has no RM (not using a dedicated tunnel), the BS just sets its maximum congestion window size to the product of the rate and its RTT.

7. The BS starts writing to the requesting UCL.

8. The UCL asks its RM for its download rate.

9. The UCL gets the download rate from its RM.

10. The UCL sets its rcvw to the product of this download rate and its flow's RTT and continues to read (download) the content.

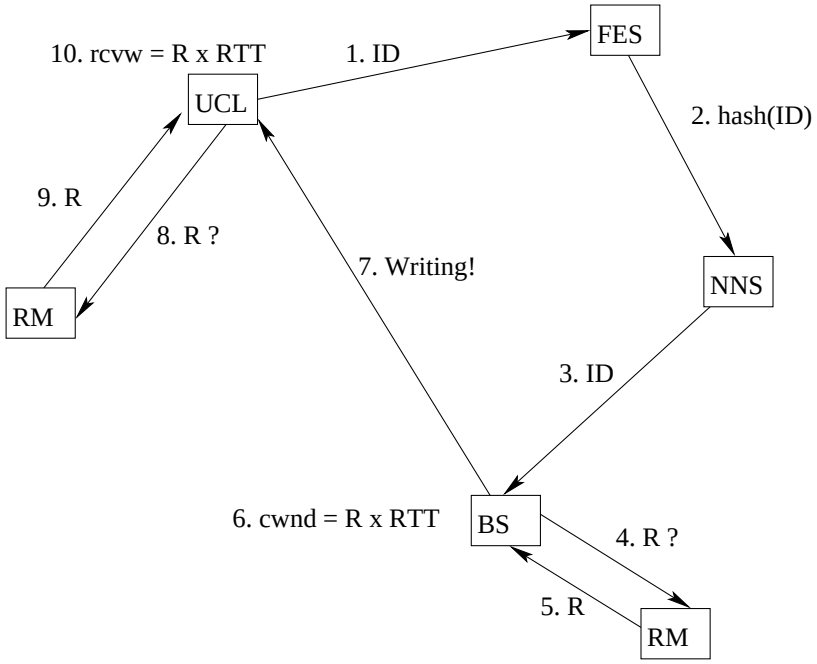

Figure 3.5: Serving External Read Request

3.8.4 Updating Rate of On-going Flows

To update the rates at which on-going flows in the cloud should send data, both the sender and the receiver have to update their windows. Suppose the lowest level parent (switch/router) both the sender and receiver share is at level h. The sender sets its *cwnd* to the product of the level h upload rate it obtains from its RM and the current RTT of the flow. Besides, the receiver sets its receive window to the product of the h level download rate it obtains from its RM and the current RTT. These two window updates in each BS are done by the RM of each BS every control interval τ.

3.9 General Network Topologies

In the above sections of this chapter, we have shown how SCDA works with tree type network topologies described by figure 3.1. The design of SCDA also applies to a general data center network topology such as figure 8 of [26] and figure 1.3. For such topologies, the RM associated with each BS computes the values of $S_{d,u}(t)$ given by equation 3.6 for each group of flows sharing the same path upto the highest level switches. To form these groups, the RMs can use their routing tables. An RM of each group sends the $S_{d,u}(t)$ values to its parent RA as shown in figure 3.6. The RA aggregates $S_{d,u}(t)$ values of groups which follow the same path starting from its associated switch. This aggregate value is then forwarded to the RA associated with the corresponding next hop interface. The control packets from each RM and RA group which carry the $S_{d,u}(t)$ values can have a destination IP prefix or IP address which matches their group (path). This destination IP prefix or IP address can be the IP address of the highest level datacenter switch/router in the path of the group. This way (using the source and destination IP of the control packets), the parent RA can detect to which interface a control packet carrying the $S_{d,u}(t)$ value belongs.

The routing tables can be calculated by each RM and RA (distributed). They can also be obtained by a central agent (controller) and shared among all RMs and RAs. To form a topology for route computation, each RM and RA share the weights of the links they represent. This can be done using message passing (inter-process communication) if the RMs and RAs are located in the same server system. Each RM and RA can also send the weights of the links they represent to a central server (controller) which forms the topology with link weights from which the shortest paths are computed. Each RM needs to form the groups from the routing table. The RAs however do not necessarily have to have the routing tables. They only need the forwarding tables, to forward the control packets of each group to the corresponding next hop.

This scheme can apply to general network topologies such as the one shown at figure 3.7 with the following steps.

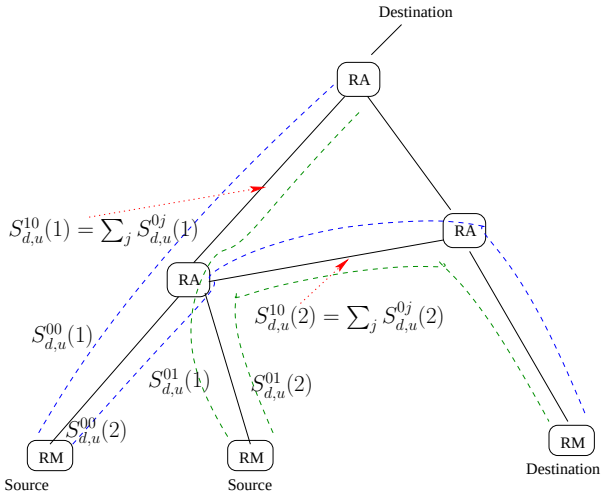

Figure 3.6: The SCDA Max/Min Scheme With Multiple Possible Paths per Server

- Each RM forms the groups based on the paths they follow obtained from its routing table.
- Each RM sends control packets carrying $S_{d,u}(t)$ of each group to the corresponding parent RA. The control packet carries a common prefix of the group it represents. The destination IP address of each control packet may be the IP address of the highest level datacenter switch/router in the path of the group.
- Each parent RA aggregates these values of the corresponding interface and obtains the rate $R_{d,u}(t)$ values for each interface. RA uses the source and destination of the control packets to identify which packet belongs to which interface.
- The rate $R_{d,u}(t)$ values can then be used as link weights of their corresponding interface.

The RAs and RMs can be co-located in the same server system. Figure 3.7 is meant to show that there are RMs and RAs associated with each source and destination and with each switch/router respectively.

The weight of each link is the value of $R_{d,u}(t)$ of that link given by equations 3.2 or 3.5. In this case, a max/min algorithm has to be used to find the best path and the rate in that path. This is done by first finding the minimum rate of each path and then taking the path with the maximum such rate as shown in [39] and in Chapter 2 for QCP.

3.10 Experimental Results

We implemented SCDA in the NS2 simulation package. We use the network topology described by figure 3.8. We run experiments using content size and flow arrival rate traces and well known distributions. In the

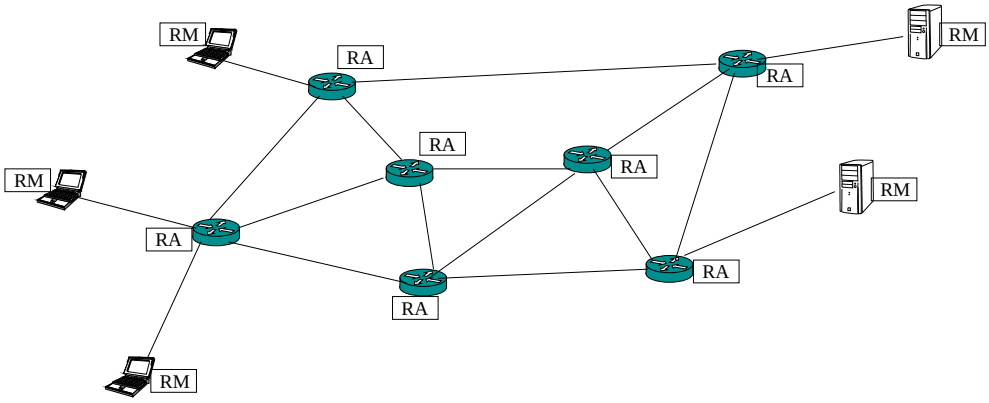

Figure 3.7: RMs and RAs Used with General Network Topolgies

experiments we show how SCDA compares with RandTCP, a random server selection and TCP rate control approach used by well known architectures such as VL2 [25] and Hedera [26]. We next discuss our initial experimental results.

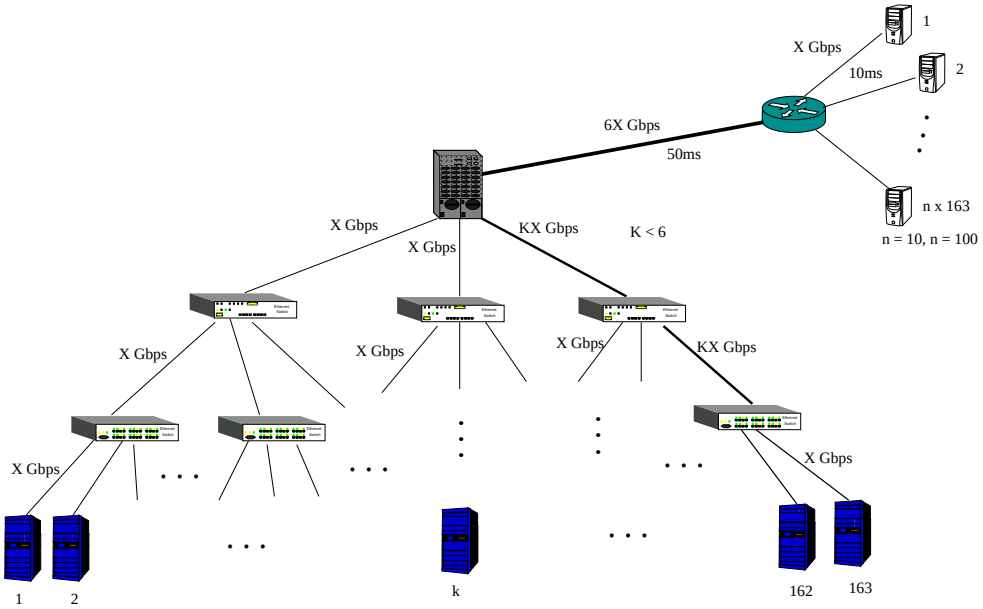

Figure 3.8: SCDA experimental network topology: Propagation delay of each local datacenter link is 10 μs

3.10.1 Using File Size and Flow Arrival Traces

In this section we will first discuss experimental results based on CDN YouTube video traces and then using general datacenter traffic traces.

Video Traces

For the first group of experiments we use CDN traces for the file sizes [76] and flow arrival rates [77]. The file size traces belong to control flows which are less than 5KB and YouTube video flows which are greater than or equal to 5KB. A bandwidth factor of $K = 3$ is used for these experiments. By varying this bandwidth multiplier of some links in the right side of the topology given in figure 3.8, we show that SCDA is not restricted to equal bandwidth datacenter architectures. We calculate arrival rates to 20 of the 2138 YouTube servers considered in [76] proportionally to scale our simulation. For these set of experiments we use the base bandwidth of $X = 500\,Mbps = 0.5Gbps$.

The first set of video trace experiments includes both the control and video flows. The control flows are HTTP messages exchanged between the Flash Plugin and a content server before a video flow starts. Figure 3.9 shows that SCDA achieves higher average instantaneous throughput than RandTCP based schemes. Figure 3.10 shows that most of SCDA flows finished in a much shorter time when compared with RandTCP based schemes. A combination of random server selection and TCP behavior causes the performance decline of RandTCP based approaches. Figure 3.11 also shows that SCDA can achieve a smaller AFCT (average file completion time). AFCT of flows of some size is obtained by taking the average completion times of all flows with that size which finish within simulation time.

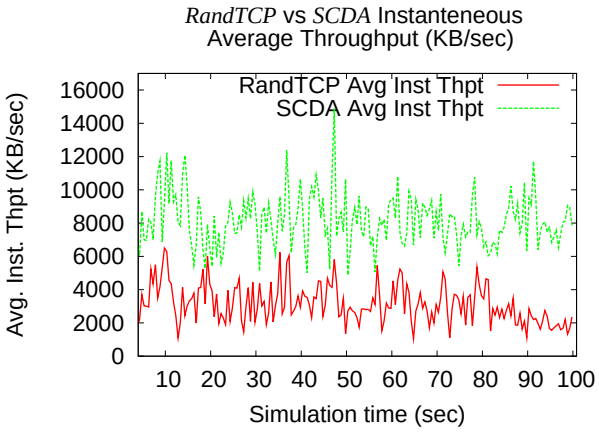

Figure 3.9: Instantaneous throughput comparison of SCDA and RandTCP based: Using Video Traces with control flows

We have also conducted trace based experiments excluding the video control flows (only YouTube video flows) as shown in figures 3.12,3.13 and 3.14.

From figures 3.9 and 3.12 it can be seen that upto 50% higher average throughput than RandTCP based schemes can be obtained. Figures 3.10 and 3.13 as well as figures 3.11 and 3.14 show that SCDA can result in FCT (file completion time) which is more than 50% lower than that of RandTCP based schemes. As

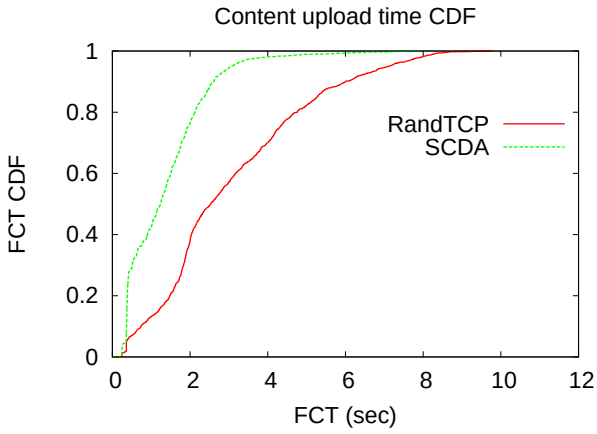

Figure 3.10: FCT CDF comparison of SCDA and RandTCP based: Using Video Traces with control flows

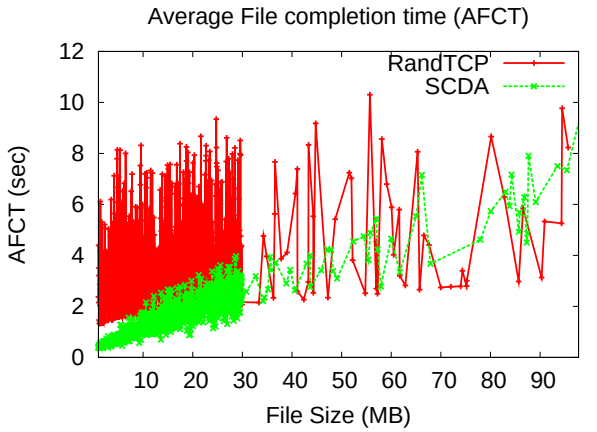

Figure 3.11: AFCT comparison of SCDA and RandTCP based: Using Video Traces with control flows

shown in [76, 78], there is a maximum size limit of about 30MB for most YouTube video files. The AFCT in figures 3.11 and 3.14 show that the transfer (upload or download) time of these files is more than 60% smaller than RandTCP based schemes for the topology given in figure 3.8. The transfer times of the very few files which are larger than 30MB is also not larger than that of the RandTCP based schemes. The wild fluctuations of the AFCT of the RandTCP based schemes is because of the random server selection and the behavior of TCP in not knowing the appropriate sending rate. On the other hand SCDA gets explicit bottleneck rate share information of each flow from the interactions of the RM (resource monitors) and RA (resource allocators).

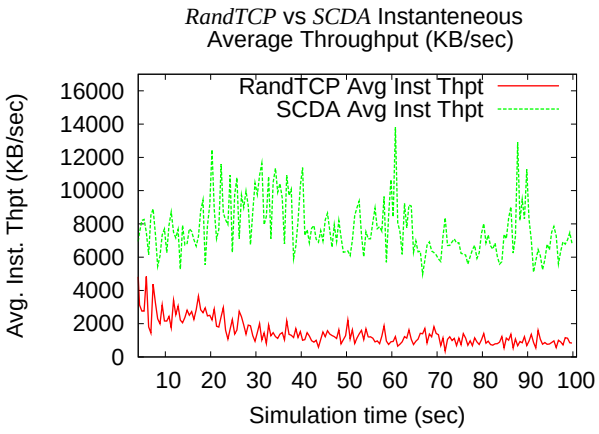

Figure 3.12: Average Instantaneous Throughput comparison of SCDA and RandTCP based: Using Video Traces without control flows

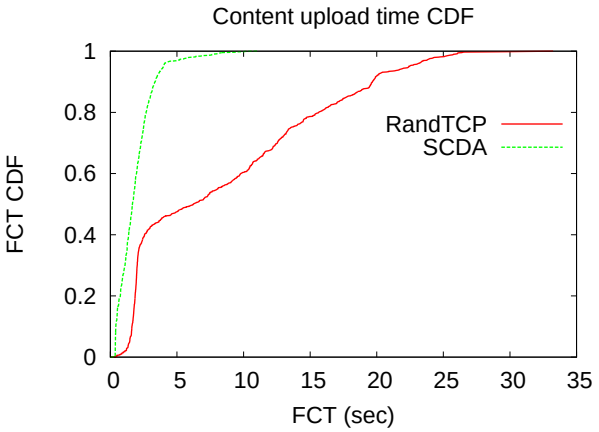

Figure 3.13: FCT CDF comparison of SCDA and RandTCP based: Using Video Traces without control flows

General Datacenter Traces

We have also evaluated the performance of SCDA using datacenter file size and flow inter-arrival traces obtained from [25] and [5] respectively. For the first set of experiments shown in figures 3.15 and 3.16 we use a bandwidth factor of $K = 1$. Similar to the plots in section 3.10.1, these plots also show that SCDA achieves a FCT which is upto 50% lower than RandTCP based schemes.

For the second set of experiments shown in figures 3.17 and 3.18 we use $K = 3$. As can be seen from figures 3.15 and 3.17, the AFCT of RandTCP shows some wild fluctuations as a result of random server selection and TCP behavior. The random server selection may result in assigning flows (requests) to servers which are congested with long-lived (elephant) flows. As a result AFCT of SCDA is upto 50% lower than that of the RandTCP based schemes. The CDF figures 3.16 and 3.18 also show that more than 60% of

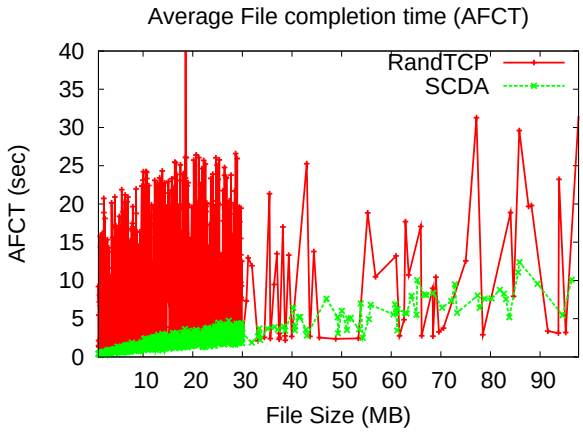

Figure 3.14: AFCT comparison of SCDA and RandTCP based: Using Video Traces without control flows

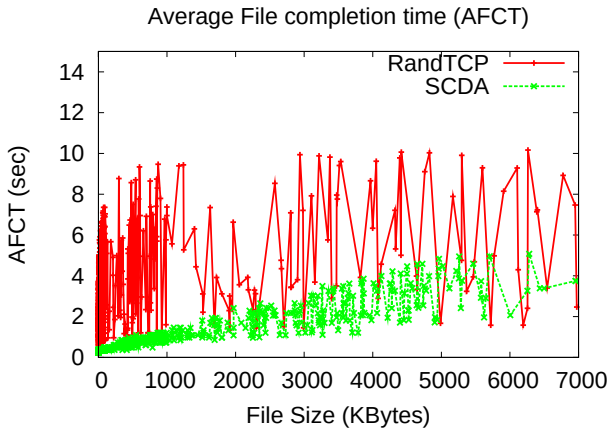

Figure 3.15: AFCT comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 1$

SCDA flows achieve upto 50% smaller transfer times than RandTCP based approaches.

3.10.2 Using Pareto File Size and Poisson Flow Arrival Distributions

We have also used Pareto distribution to generate the file (content) sizes and Poisson distribution to generate the flow inter-arrival times. We set the base bandwidth value $X = 200\,Mbps$ and the bandwidth factor $K = 3$ for this experiments. File sizes are Pareto distributed with mean $500KB$ and shape parameter of 1.6. Flow arrival rates are Poisson distributed with mean 200 flows/sec. Consistent with the trace based plots in section 3.10.1, the distribution based figures 3.19 and 3.20 show that SCDA outperforms RandTCP based schemes.

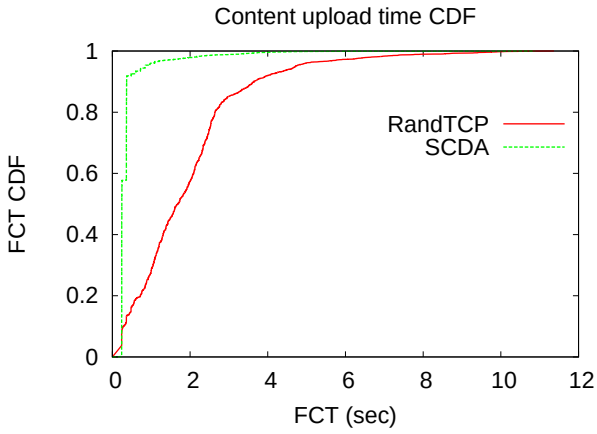

Figure 3.16: FCT CDF comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 1$

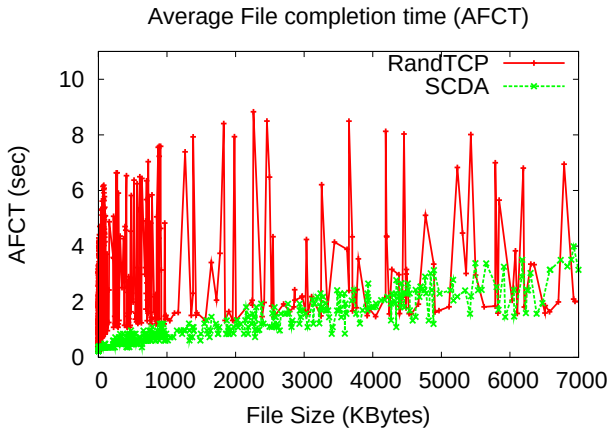

Figure 3.17: AFCT comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 3$

3.11 Related Work

In this section we discuss existing server selection and congestion control mechanisms in data center networks. As also discussed in [79], in the Fat-Tree architectures [80, 72], each switch in the lower level of the topology regularly (every second) measures the utilization of its output ports. This measurement is done at regular interval (every 10 second). If the utilization of the output ports are mismatched, the switch reassigns a minimal number of flows to the ports. Fat-Tree uses this local heuristic to balance load across multiple shortest paths. However, as discussed in [79], this heuristic results in a 23% performance gap from the optimal value resulting in possible packet losses and congestion. This demands for globally optimal decisions as also pointed out in [79]. Our SCDA scheme has an adaptive global and local view of the cloud data center to achieve optimal resource allocation.

The VL2 architecture [25] randomly chooses intermediate switches to forward flows to servers using

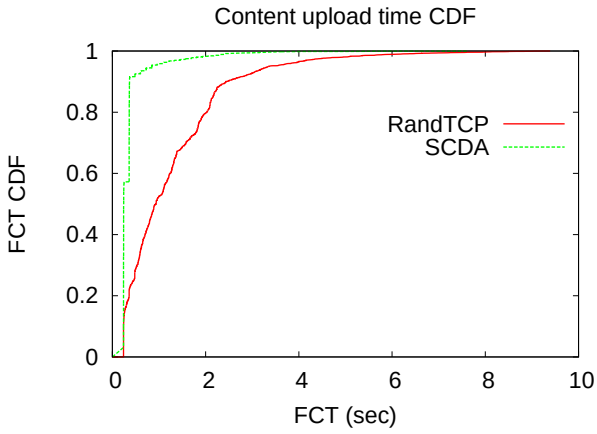

Figure 3.18: FCT CDF comparison of SCDA and RandTCP based: Using Datacenter Traces with $K = 3$

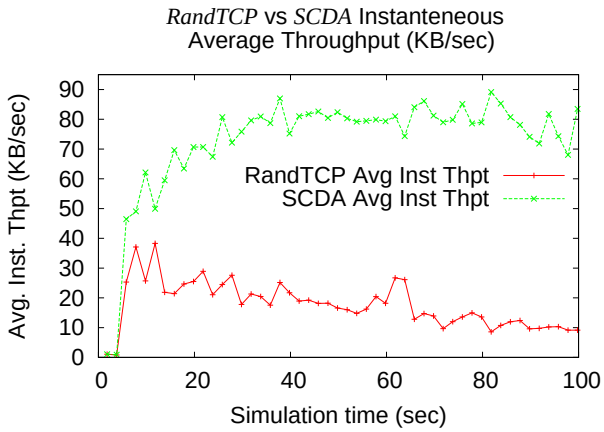

Figure 3.19: Throughput comparison of SCDA and VL2: Using Pareto and Poisson Distributions

equal-cost multi-path routing (ECMP) [81] and valiant load balancing (VLB) [82]. As also pointed out by the authors of VL2 and in [26], both ECMP and VLB schemes of random placement of flows to servers can lead to persistent congestion on some links while other links are under-utilized. This is specially the case with "elephant flows" in the network or in a network where there is multimedia video streaming. One of the reasons for this is the inability of ECMP to track and adapt to instantaneous traffic volume. It should be noted that per-flow VLB which is the case with VL2 becomes equivalent to ECMP, with both utilizing random switch and hence server selection mechanism.

The Hedera [26] flow scheduling utilizes ECMP for short-lived flows and a centralized approach to route large flows (with over 100MB of data). Leaving the complexity of classifying flows and detecting the amount of data, flows can send before they send it, aside, the work in [79] showed that the Hedera scheme performed comparable to (not better than) the ECMP as most of the contending flows had less than 100MB of data

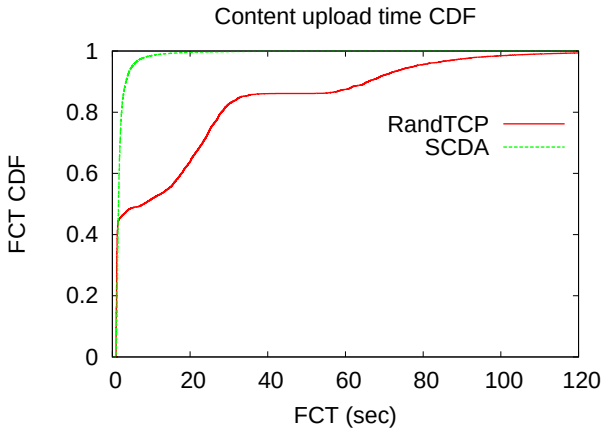

Figure 3.20: File Completion Time (FCT) comparison of SCDA and VL2: Using Pareto and Poisson Distributions

to send. Unlike VL2 and Hedera, the SCDA server selection approach adaptively takes many resource constraints into account.

A traffic engineering scheme, called MicroTE, is also proposed in [79] to address the problems of the above data center architectures. Just like our initial work [83, 84], the MicroTE approach uses a controller which aggregates network traffic information from the top-of-the-rack (ToR) switches in the data center. The controller then tries to solve a usual capacity constrained linear programming problem using a heuristic approach. The approach first sorts predictable ToR pairs which exchange traffic according to their traffic volume. The prediction is done over 2 seconds interval. Leaving the complexity involved in doing this prediction aside, for N nodes and E edges in the network, MicroTE has a computational complexity of $\bigodot(PNlog(N) + P + PlogP)$ for P predictable ToR (source-destination) pairs. Besides the link weight used for the optimization problem is the inverse of available capacity. If there are two links with the same available capacity, and if one of them has more flows sending data to it, then it is not a good idea to consider the two links the same way. This implies that inverse of available capacity is not a good link metric as it does not take into account the number of flows sharing the link. To deal with such link weight issues, SCDA uses a flow rate as a link weight. Besides, SCDA uses a distributed and adaptive scheme to implicitly solve the optimization problem.

Besides, all of the above architectures depend on TCP to determine sending rates of the flows. Among other weaknesses, TCP results in very high AFCT [8]. Clean slate congestion control protocols such as XCP [17] and RCP [8] require modifications to routers/switches and the TCP/IP stack. The SCDA architecture mitigates the drawbacks of TCP without requiring changes at the routers/switches and TCP/IP stack. All of the above schemes do not achieve max/min fairness and do not have a mechanism to detect SLA violation

in realtime. The RA and RM software components along with some mathematical formulation allows SCDA to achieve max/min fairness and to detect SLA violations without the need of hardware or TCP/IP changes.

3.12 Summary

In this chapter we have presented the design of SLA-aware cloud data center architecture (SCDA) for efficient content storage and retrieval. Current large scale distributed file systems such as the Google File System (GFS) and its derivative, the Hadoop File System (HDFS), rely on a single name node server (NNS) to manage metadata information of all chunks stored in all block (chunk) servers (BS) in the cloud. This design can make GFS and HDFS bottlenecked at the single NNS. The design of SCDA solves this problem by introducing a light weight front end server (FES) which forwards requests to multiple NNS.

Existing schemes such as GFS, HDFS, VL2, Hedera rely on TCP to avoid congestion and determine the sending rates of flows. The SCDA architecture uses efficient congestion control and server selection mechanism to decide *where in the cloud (distributed system)* to store data and *at what rate to transmit* data. This design enables SCDA to efficiently balance load among all data and name node servers automatically. The resource monitor (RM) and resource allocator (RA) components of SCDA also allow SCDA to be implemented without the need to change network switches, routers and the TCP/IP packet header format.

The design of SCDA has other important features. It can adaptively achieve max/min fairness which is described to be NP hard in the current literature. It can automatically detect SLA violation in realtime. It is scalable as all its components can run independently by exchanging messages. The design of SCDA is extended to be more energy efficient. SCDA can serve as a multi resource allocation scheme where the bottleneck resource can be other than the link bandwidth. The prioritized allocation mechanism of SCDA allows it to easily make QoS rate assignments.

We have implemented SCDA in the NS2 simulator [63] and compared it against well known existing schemes using random server selection and TCP (RandTCP). Simulation results show how SCDA outperforms RandTCP based approaches such as VL2 and Hedera in terms of content transfer time and throughput.

Chapter 4

Hincent: Cross-Layer Quick Content Distribution With Priorities and High Incentives

4.1 Introduction

In the previous Chapter 3 we presented a cross-layer SCDA scheme for cloud datacenter networks. In this chapter we present *Hincent*, an efficient *prioritized rate allocation (congestion control) and content source selection (content routing)* protocol offering *high and fair* incentives to participating peers. There can be multiple content sources (peers). Among these sources, *Hincent* selects the one which offers high throughput path to the destination. This makes *Hincent* cross-layer routing and congestion control protocol.

The design of *Hincent* enables distributed network peers to securely exchange content by providing high monetary and bandwidth incentives for their resource (bandwidth, storage, energy, processing, etc) used in the content transfer. It allows users to have full control of their contents which can be a 2D, 3D data or ordinary file. *Hincent* can limit the lifetime of the content to a user-defined parameter. This content age and the prioritized rate allocation features of *Hincent* are specially important for 2D and 3D live streaming contents which have real time requirements. For instance, to render a 3D video, streams should be synchronized and rendered within a short time gap between them. The fair and accurate incentive, rate allocation, enforcement and content source selection mechanisms of *Hincent* allows peers to exchange content with smaller transfer time than existing schemes. The *Hincent* protocol does not need changes to the TCP/IP stack and existing network devices (routers, switches) that it can be easily deployed in the current Internet.

We also discuss an extension of *Hincent* using surrogate cloud or cloudlet servers to help peer clients transfer contents faster than using existing schemes. The servers are equipped with OpenFlow vSwitches and form a network. These servers in the network are connected using either dedicated or overlay links. Cloudlets [10] are decentralized and widely-dispersed Internet infrastructure whose compute cycles and storage resources can be leveraged by nearby mobile computers.

We have implemented *Hincent* in the NS2 [63] simulator and using an Apache SQL server with PHP in Linux virtual machines. The NS2 simulator is so robust that descriptions of the streams of the 3D content

can be taken as inputs to produce an emulated 3D video as output. The simulation results show how *Hincent* can outperform existing well known content distribution schemes in terms of download time and throughput. The results also demonstrate that the different components of *Hincent* work according to the design. The SQL implementation of *Hincent* using PHP shows that *Hincent* can scale to millions of peers and contents.

In [41] we presented a short version of our *Hincent* work. In that short version, among other things, we did not show how *Hincent* rate allocation can be TCP friendly, there was no discussion of multiple server selection policies and there was no extension of *Hincent* using surrogate servers. Besides, the short version did not discuss the *Hincent* content index management (CIM) schemes and how the schemes scale. We also did not present Apache SQL server implementation experiments of *Hincent* CIM in our short version.

The main contributions of this work are as follows.

- We have designed an efficient content distribution protocol (*Hincent*) with cross-layer content routing (content source selection) and congestion control mechanisms. It can allow distributed users (peers) to have full control of their contents while securely sharing them.
- We have shown that *Hincent* provides accurate and efficient incentive mechanisms to benefit content providers, content users and network operators. The incentive is in real monetary values (*monetary incentive* mode) and can also be translated into download rate (*bandwidth incentive* mode).
- *Hincent* is a max/min protocol making efficient utilization of network resources resulting in high throughput and lower transfer time.
- The prioritized rate allocation mechanism of *Hincent* allows some applications such as multi-view 3D streaming to assign higher rate to some flows (streams). The design has content lifetime feature to ensure efficient transmission of live and multi-view content.
- *Hincent* uses an efficient content index management scheme making it deployable in current networks without having to change the TCP/IP stack, routers or switches.
- We have presented an efficient algorithm which extends *Hincent* to use surrogate servers to help peers transfer contents faster.
- We have implemented *Hincent* in the NS2 simulator and evaluated its performance. Results show that it can achieve on average about 30% lower content transfer time when compared with existing schemes.
- We have experimented with *Hincent* using Apache SQL server, and have shown that *Hincent* scales.

The rest of the chapter is organized as follows. In section 4.2 we present the *Hincent* protocol. In section 4.3 we present the methods *Hincent* uses to calculate the rates and prices which are used in the algorithms of the *Hincent* protocol. *Hincent* content source selection mechanism which is also used by the *Hincent* algorithms is presented in section 4.4. In section 4.5 we show how *Hincent* rate allocation is TCP friendly. Section 4.6 discusses *Hincent* scenarios when a flow ends. A list of other server selection policies is presented in section 4.7. In section 4.8 we present the content index management component of Hincent. In section 4.9 we show how Hincent content index management scales with the growth of the number of content records. Section 4.10 shows how our scheme deals with scarce backbone bandwidth. In section 4.11 we show how *Hincent* can be extended using surrogate servers to help peers exchange contents. We evaluate the performance of *Hincent* in section 4.12. Analysis of related work is given in section **??**. Finally, we give conclusion of the chapter in section 4.14.

4.2 *Hincent* Protocol

The *Hincent* protocol consists of network and content models, logical and physical architectures and algorithms described below.

4.2.1 Network and Content Model

The network model of *Hincent* consists of a graph $G = (N, E)$ of nodes N and edges E as shown in figure 4.1. The node set V consists of the CDN servers which provide content and the peers which provide and/or request for content. The edge set E consists of all edges going to and from the nodes. All nodes are linked with each other over the Internet which may consist of multiple backbone networks. Each node has link with specified upload and download capacities it buys (gets) from network operators. The operator backbone network usually has enough bandwidth to provide bandwidth guarantee to the users (nodes). This is usually done using protocols such as the OSPFv3 as a Provider Edge to Customer Edge (PE-CE) Routing Protocol [71].

The *Hincent* data model consists of content which is sent from the CDN servers or from some peers and exchanged between the peer nodes. We classify the data (contents) into none real-time ordinary static file (OSC), a realtime (live and none-live) streaming video content like 2-dimensional (2D) YouTube or a 3-dimensional (3D) video content [85]. The 3D Tele-Imersive content involves multiple streams from different view angles which have to be synchronized by the receiving end to produce a 3D multi-view streaming video. To synchronize the contents, *Hincent* uses content lifetime threshold based on how long a receiving node can buffer. For a stringent 3D Tele-Imersive environment, where the peers have to produce interactive content,

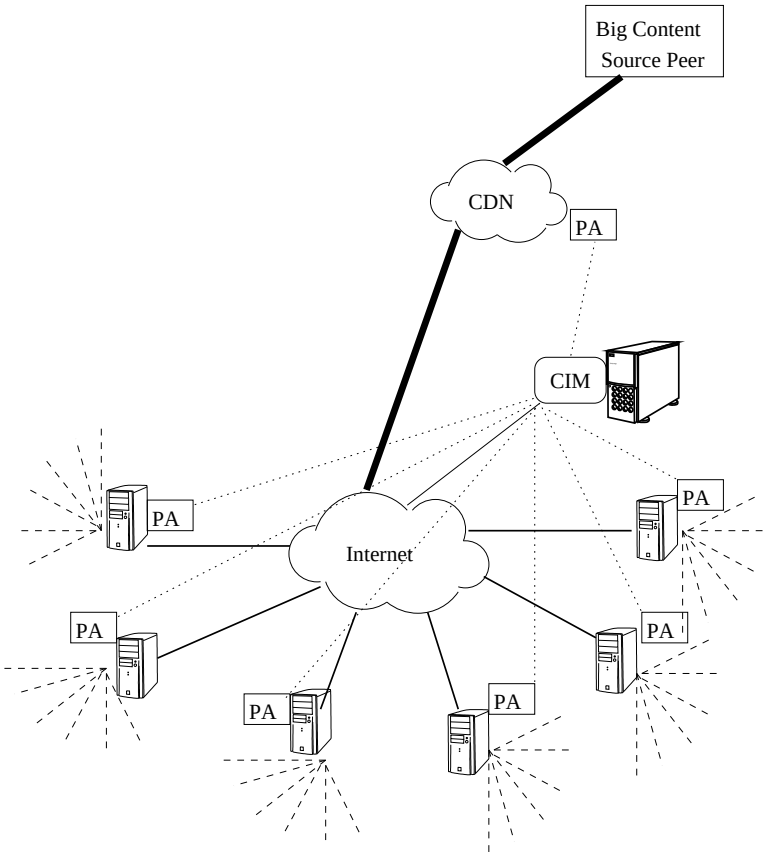

Figure 4.1: The *Hincent* Architecture

the content lifetime becomes very small to ensure a very small delay. For most cases where nodes view the 3D content, the content life time can be relaxed.

4.2.2 Logical and Physical Architectures

The *Hincent* architecture aims to efficiently distribute content to network peers benefiting all content actors (content providers, content users and network operators). As shown in figure 4.2, it consists of a content information (index) manager (CIM) and peer agent (PA). A PA connects a peer with the CIM. A CIM registers peers and chooses content source to requesting peers. The CIM is made up of the light weight front end server (FES), content information database (CID), the complaint manager (CM) and the archive manager (AM). The CID consists of a database of contents information such as the source peers, source upload rates. The CM manages reports about misbehaving peers. The AM manages old content information and transaction logs to perform offline content index analysis. The FES forwards requests to register a new peer, a new content, or requests for a content, to the respective CID tables. The FES also forwards peer complaints to the CM. The CM contacts the AM for complaint history. The CIM archives old content state

information at the AM.

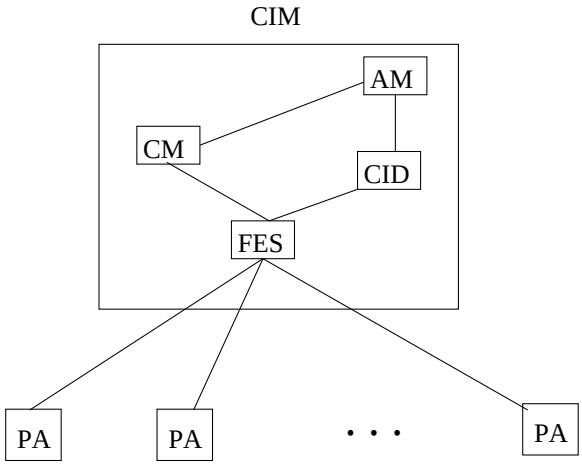

Figure 4.2: The *Hincent* Logical Architecture

The *Hincent* physical architecture can be described by figure 4.1. The architecture consists of the peer nodes with their PA, the CIM and a big content source peer connected to its CDN with a bigger link. The big content source sends its content to the content distribution network (CDN) which informs the CIM of the new content. The content source which can be any peer with a PA can also inform the CIM of its content directly. The other peer nodes can then send a content request to the content information manager (CIM) via their peer agent (PA). The peers can get the content either from the CDN or other peers whichever gives the highest throughput to price ratio as discussed in the next section 4.4.

Hence *Hincent* consists of 3 main logical parts namely *content index manager* (CIM), *prioritized max/min rate allocation* (PRA) and *bandwidth and content pricing* (BCP) as shown in figure 4.3. These components interact with each other. The CIM consists of databases with information of peers and data contents. The PRA component is done with the help of the CIM and distributed peer agents (PA). It is where prioritized rate is calculated for each upload and download link of the peers and other main content servers. The rates are then used to choose a content source and to set the sending rates of the corresponding flows. BCP which is also done by the CIM and PA is a component where the bandwidth and content prices are calculated adaptively to ensure incentives between the participating peers. Peers which upload more, earn more credit which can be of monetary value or in terms of download bandwidth or content discounting.

We next discuss the *Hincent* algorithm involving the CIM and PA.

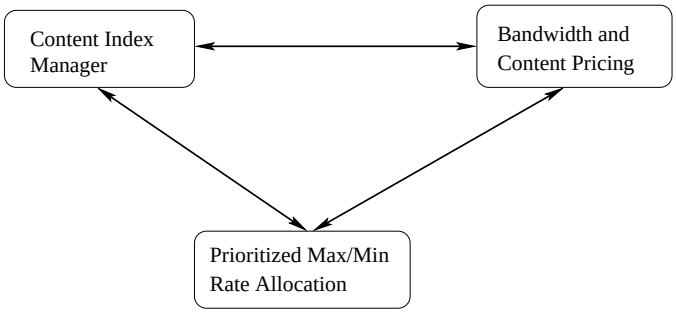

Figure 4.3: *Hincent* Logical Components

4.2.3 *Hincent* Algorithms

The *Hincent* algorithms are cooperatively run by the CIM and PA, to compute transmission rate and price (bandwidth, content) metrics for the content distribution. To obtain the rates at which each content is transmitted from one node to another node and the bandwidth usage price, *Hincent* first carries out temporary rate and price calculations at the CIM at every request or at every control interval τ. The rates and prices are then sent to the PAs, updated by the PA and sent back to the CIM. The CIM then uses these rate and price values to select a content source (peer or CDN server) and determine the rate at which content is transmitted.

To define the *Hincent* rate and price metrics, we first present the following notations in table 4.1. For each *Hincent* parameter $X \in \{R, C, Q, \hat{N}, N, n^j, R^j, M^j, p, \wp^j\}$, with j being a flow index, described in the table we use the notation,

$$X_{d,u} = \begin{cases} X_d & \text{if } X \text{ is a downlink } \textit{Hincent} \text{ parameter,} \\ X_u & \text{if } X \text{ is a uplink } \textit{Hincent} \text{ parameter.} \end{cases} \tag{4.1}$$

Table 4.1: *Hincent* Parameters

Variables	Description
$C_{d,u}$	Link capacity
τ	Control interval
$R_{d,u}(t)$	Base link rate allocation of the current interval (round)
$N_{d,u}(t)$	Number of flows in the link during the current round
$R^j_{d,u}(t)$	Link rate allocation of flow j for the current round
$M^j_{d,u}$	Minimum rate requirement of content flow j
$p_{d,u}(t)$	Per packet price
$\wp^j_{d,u}$	Priority weight of flow (stream or chunk) j

With the above notations, the *Hincent* algorithm consists of the following steps.

Initialization steps:

- In the *Hincent* deployment scenario each peer sets up a personal web (content) server (with the help of *Hincent*). The web-server can be hosted at a home server, a friend server, an ISP or a cloud.
- Each participating peer and CDN server first initialize their up link and down link base rates to the uplink and downlink capacities, they dedicate to the *Hincent* system.
- Each participating peer and CDN server also initialize their unit per packet price (bandwidth) to some value. In this study, the CIM sets the initial per packet bandwidth prices of the peers to a small fraction of real CDN bandwidth prices used by the Amazon CLoudFront [86]. Even though we consider only bandwidth price in this chapter, the price may include other costs such as peer storage, energy, processing, content cost and other costs.
- Each participating peer and CDN server with a content then send these rate and price values along with other peer and content fields such as peer ID and content ID to the CIM.
- CIM authenticates and registers the requesting peers and the content sources.

Content request steps:

- Peer which is interested in a specific content sends (via its PA) a content request along with minimum rate requirement, M_u^i to the CIM. The most popular content information can be displayed by the CIM for other peers to see. Peers can also lookup the content from the CIM tables (via a web interface).
- If no peer has the desired content, the CIM sends the IP address of a CDN (cloud) server which has the content to the requesting peer and the IP address of the requesting peer to the selected CDN server. The CIM can also use existing search engines such as Bing and Google to look for the requested content. Once a requesting peer finds and clicks at the requested content, the index of the content can be stored as being available in the requesting peer by the CIM. Next time other peers request the CIM for the same content, the content can be directly served from the peer which got the content from the search engines. It is important that the CIM and the PA save the link to the original source of the content. This helps the PA to update the content and attract more customers with up-to-date content. Additional *Hincent* content servers can also keep a copy of the searched content and its original link to provide fresh content to peers and to monitor if the content source peers are offering fresh content. Peers have incentive to maintain fresh content as doing so attracts more customers (other peers).
- If there is (are) other peers which have a content requested by another peer, the CIM chooses the node (peer or CDN server) which gives the best metric (low price, high throughput) based on the content source selection policies discussed in section 4.4.

After the content request is received by the CIM, CIM and PA update steps are carried out before content transfer to avoid resource congestion and to achieve max/min resource (link) usage respectively.

CIM update steps:

- To reserve a minimum bandwidth requirement for the requesting peer, CIM subtracts M_u^i of request i from the remaining uplink capacity of the content source and M_d^i from the remaining downlink capacity of destination peer. This involves only a single subtraction operation. This remaining capacity is used in equation 4.5 of the rate calculation. If either of the remaining bandwidths is negative, the CIM informs the requesting peer that its request cannot be fulfilled.

- CIM increments the flow priority weight sum to be used in equation 4.5. This involves one addition instruction. The flow priorities are globally known to the CIM or specified by each requesting peer. The PA and the CIM then calculate the corresponding weights of the priorities.

- After accumulating the remaining bandwidth values and the sum of the priorities used in equation 4.5, the calculations of the base rate using equation 4.5 and price values using equation 4.7 can be done periodically to further reduce more computational overhead.

- CIM sends the IP of the selected content source along with the base upload rate $R_u(t)$ and the *contentHash* of the requested content to the requesting peer. The *contentHash* is to check for content integrity.

- CIM sends the base download rate $R_d(t)$ of the requesting node to the selected source.

When a PAs of the content source and destination receive the rates $R_{d,u}$ of their uplink and downlink flows from its CIM, they performs the following.

PA update steps:

- Use the uplink and downlink rate values of each of the flows of its node received from its CIM to obtain the effective flow count for all uplink and downlink flows of its node using equations 4.9 and 4.8.

- Calculate new rate values using the effective flow count as given by equation 4.10. This new rate ensures that a capacity unused by some flows is being used by other flows making *Hincent* a max-min fair algorithm. This is because some uplink flows may be bottlenecked at the downlink and vice-versa.

- Calculate the new price value based on the new rate values using equation 4.7.

- Send the new base rate values obtained using equation 4.10 back to the CIM. The new price values can also be sent to the CIM saving the CIM some computational costs.

The CIM then calculates its new price values and uses both the new rate and price values to select content sources (peers or CDN servers) for each request for content.

Rate enforcing and content download steps:

- Both content source and destination calculate the new rate $R^i_{d,u}$ values of each of their uplink and downlink flows (streams) i using equation 4.11.
- Both content source and destination enforce the rate allocation as follows. First the destination node sets its receive window w^i_r of flow i as

$$w^i_r = R^i_d(t)RTT^i. \tag{4.2}$$

Then the corresponding source of the flow (stream) i sets its congestion window w_i as

$$w_i = min(w^i_r, R^i_u(t)RTT^i). \tag{4.3}$$

If the bottleneck link is somewhere in the Internet which is described as "Internet" node in figure 4.1, then the destination of flow i sets its receive window size as given by equation 4.2. And the source of flow i sets its maximum congestion window size w^i_M as

$$w^i_M = R^i_u(t)RTT^i. \tag{4.4}$$

Such a backbone bottleneck scenario can be detected by multiple packet losses after *Hincent* allocation, though we do not expect such a scenario to happen as discussed in section 4.2.1. More on this will be discussed in section 4.10.

- Requesting peer downloads the content from the source whose IP address it got from the CIM.

Price enforcing steps:

- Requesting peer via its PA asks for the *contentOldKey* from the CIM (CID) to decrypt the content it downloaded.
- The CIM increases the total amount $\ddot{E}$ of credit, the content source earns, and the total amount $\ddot{P}$, the receiving peer pays, each by the $contentSize \times p_{d,u}(t)$. *contentSize* is in packets.

- The CIM charges the requesting peer the specified amount and checks if the peer's balance has not fallen negative.
- If the requesting peer has enough credit (has paid for the content download), the CIM sends the *contentOldKey* to it (the peer). Otherwise the peer cannot decrypt the content after wasting its bandwidth.
- If the peer gets the decryption key, the CIM records the *contentID* of the downloaded content as available at the requesting peer unless the peer indicates it does not want to share the content. The efficient incentive mechanism of our protocol encourages peers to share contents.
- At the CIM when the flow of the requesting peer finishes (downloading the content), the remaining uplink bandwidth of the content source and the remaining downlink bandwidth of the receiving peer are increased by the minimum rate requirement of the flow which finished and the respective priority weights sums decrease by the priority weight of the flow which finished. CIM then updates the rates and prices using equations 4.5 and 4.7.

We next show how the *Hincent* rate and price are calculated.

4.3 *Hincent* Rate and Price Calculation

The temporary down-link (d) and up-link (u) rates of every node (peer or CDN server) are calculated by the CIM as

$$R_{d,u}(t) = \frac{C_{d,u} - \sum_{j}^{N_{d,u}} M_{d,u}^{j}}{\sum_{j}^{N_{d,u}} \wp_{d,u}^{j}} \tag{4.5}$$

where the notations are described in table 4.1 and $\wp_{d,u}^{j}$ is the priority weight of request j. If all requests have the same normalized priority weight, $\wp_{d,u}^{j}$, then $\sum_{j}^{N_{d,u}} \wp_{d,u}^{j} = N_{d,u}$.

The temporary uplink and downlink rates R_u^i and R_d^i of flow i are given by

$$R_{d,u}^{i} = M_{d,u}^{i} + \wp^{i} R_{d,u}(t). \tag{4.6}$$

The temporary per packet prices for the uplink (u) and downlink (d) are calculated as

$$p_{d,u}(t) = \frac{p_{d,u}(t-d) \times R_{d,u}(t-\tau)}{R_{d,u}(t)} \tag{4.7}$$

where the notations are also described in table 4.1.

When a request for content is made, the temporary rate and price calculations ensure that the CIM does not result in assigning requests to peers they do not have enough resources for. CIM leaves the refined distributed rate and price calculations to the peers.

With the temporary uplink rate of a flow k from a content source as R_u^k and the temporary downlink rate of the flow to the destination by R_d^k both obtained using equation 4.6, if $R_u^k > R_d^k$, then the content source of flow k should not send at the rate of R_u^k for flow k as it is bottlenecked in the last link to the destination. On the other hand if $R_u^k < R_d^k$, the destination node cannot receive (download) at the rate of R_d^k for the flow k. In these cases, other flows sharing the links with flow k should be able to use the corresponding uplink or downlink bandwidth unused by flow k to ensure that *Hincent* is max-min fair. To do this, some flows which cannot use the bandwidth allocated to them are counted as partial flows or fraction of a flow. We call such a count of a flow *an effective flow count*. The effective flow count of flow k at the source node is given by

$$n_u^k = \begin{cases} \frac{R_u^k}{R_d^k} & \text{if } R_d^k > R_u^k, \\ 1 & \text{otherwise.} \end{cases} \tag{4.8}$$

The effective flow count of flow k at the destination node is given by

$$n_d^k = \begin{cases} \frac{R_d^k}{R_u^k} & \text{if } R_u^k > R_d^k, \\ 1 & \text{otherwise.} \end{cases} \tag{4.9}$$

Each PA then obtains new uplink and downlink base rate values as

$$R_{d,u}(t) = \frac{C_{d,u} - \sum_j^{N_{d,u}} M_{d,u}^j}{\sum_j^{N_{d,u}} \wp_{d,u}^j n_{d,u}^j}. \tag{4.10}$$

The new per packet prices for the uplink and downlink of a node are then obtained using equation 4.7. Besides, a node resets the up and downlink rates of each of its' flow i as

$$R_{d,u}^i = M_{d,u}^i + n_{d,u}^i \wp^i R_{d,u}(t). \tag{4.11}$$

Equivalently, the uplink rate R_u^i of the flow i at a node can also be calculated as

$$R_u^i = M_u^i + n_u^i \wp^i R_u(t). \tag{4.12}$$

So far we have considered the *monetary incentive* mode of *Hincent*. The monetary incentive can also

be converted to a upload *bandwidth incentive* using the ratio of the total amount to pay to the total credit earned. To do this, the CIM informs the content source to rate-limit the requesting peer at a base rate of

$$\ddot{R} = \ddot{w}(\ddot{E}, \ddot{P}) \times min(R_d(t), R_u(t))$$

where $\ddot{w}(\ddot{E}, \ddot{P})$ is the weight function of the total monetary amount $\ddot{E}$ the requesting peer has earned and the total amount $\ddot{P}$ the peer has to pay. The *min* is a minimum function. In this study we set

$$\ddot{w}(\ddot{E}, \ddot{P}) = \frac{\ddot{E}}{\ddot{P}}. \tag{4.13}$$

Other pricing and weight functions can also be used in *Hincent*. The new weights $\tilde{\wp}_u^j$ of every request j from the requesting peer is then set as $\tilde{\wp}_u^j = \wp_u^j \ddot{w}(\ddot{E}, \ddot{P})$. This new weight is the product of the peer weight, $\ddot{w}(\ddot{E}, \ddot{P})$, and the flow (stream) priority weight, $\wp_u^j$. CIM obtains the rate allocation of the request j made by the peer as $\ddot{R}^j = M_u^j + \tilde{\wp}_u^j \ddot{R}$.

4.4 Content Source Selection

Once the CIM receives the new rate values from each PA, it obtains the new price values using equation 4.7. Then a content source for the requesting peer is selected based on the policy discussed below.

4.4.1 Highest Rate to Price Ratio Policy (HRPR)

In this HRPR policy, the CIM keeps the ratio

$$K_{d,u}(t) = R_{d,u}(t)/p_{d,u}(t) \tag{4.14}$$

of the rates to their respective prices in its peer table. When a node requests for a content, the CIM chooses a content source which gives the highest value of $K_{d,u}(t)$. This approach enables the CIM to choose a node which gives the highest rate with the lowest price. This policy takes locality into account, serving requests using local sources which give the HRPR. It can also be applied to social groups, selecting the best (with HRPR) content sources in the group for requesting peers.

4.5 *Hincent* is TCP friendly

In this section we discuss how *Hincent* deals with TCP friendliness.

Theorem 1 *A Hincent rate allocation of a flow which is not bottlenecked at a link l is TCP friendly to all flows sharing link l.*

Proof 1 *If a flow i is not bottlenecked at link l, it cannot congest link l regardless of how much its sending rate increases. This is because the flow i has another bottleneck which limits its sending rate. This in turn means that TCP flows sharing link l with flow i have enough bandwidth at link l to use. This implies that TCP fairness is not an issue at link l and flow i is TCP friendly.*

Even though *Hincent* handles scenarios where the bottleneck link can be somewhere in the backbone network, the bottleneck link in the *Hincent* architecture is usually going to be at the last mile links to and from the peers. This is because (1) users (peers) usually buy a guaranteed bandwidth and (2) the peers which can use a specific peer as a source of their content are usually scattered over a wide area each using different paths in the backbone network. Hence, if the *Hincent* flows are not bottlenecked at a link which they share with TCP, then they are TCP friendly based on the above theorem 1. In a scenario where the bottleneck link is in the backbone network, the *Hincent* flows will drop or delay packets. This congestion signal can be detected by the PA of each peer which counts the number of successfully transmitted packets. The PA compares this count over a time interval against the minimum of the uplink and downlink rates of the flow. If the PA finds that the backbone network is congested, it uses the maximum congestion window and receive window to enforce the rate allocations as discussed in section 4.10.

4.6 When a Flow Ends

When a peer wants to end a flow (stream) due to for instance 3D view change, the node sends the contentID of the flow (stream) it needs to end. The CIM then finds the corresponding global contentID in its content table, removes the contentID and releases the associated resources. It then updates the corresponding content source and destination rate and price values. The CIM also finds a new content source to all other peers which are actively downloading the content from the peer which wants to end it. Here, the CIM uses the original content source as the new content source for the peers which are using the content whose source is ending it. This is because if a new peer (which is not the original content source) is chosen to be the new source of the content, it is difficult to find (trace) out whether one of the parents (ancestors) of this chosen peer is the peer which is ending the content or not.

4.7 Other Server Selection Policies

In this section we discuss server selection policies other than the HRPR policy discussed in section 4.4.1.

4.7.1 Highest Rate Policy

In this policy the CIM selects a content source which provides the highest rate to each node irrespective of the price. So a node which is allowed to download from a content source with the highest rate pays the corresponding price. The nodes can also earn credit by allowing other nodes to download from them and then get a service whose price is equivalent to the credit they have earned as discussed in section 4.3.

4.7.2 Best Rate Fit Policy

When a node requests the CIM for a content, the CIM can also choose a content source whose upload rate is the smallest value greater than the download rate of the requesting node. This approach allows the CIM to do a best fit allocation to allow big upload requests.

4.7.3 Smallest Price Policy

If a node which requests for a content doesn't want to pay more or doesn't want to spend more of its credit, it can request a smallest price policy. In this case, the CIM chooses a content source with an upload value of at least as much as the minimum required rate for the content and with the smallest price.

4.7.4 Lowest Latency (Local Network) Policy

A user's request may have some latency constraints. In this case a user may request a node with the shortest latency. To deal with this scenario, we group peers with similar IP prefixes together. This can be done by hashing the most-significant bit-group in the IP address of the content request packets of the registering peers. We can then have one CIM responsible for each group of users (peer nodes) forming a hierarchical structure of content information managers as shown in figure 4.4. This policy can have a significant advantage in reducing backbone network link congestion as many requests can be served locally. This is another benefit to network operators. Besides, users in the same geographical location may tend to have interest to the same content making it easy for the content source selection algorithm to decide.

To use this policy, users send a request to the FES of the CIM which then hashes the requester's IP prefix values and forwards them to their respective CID tables. This approach also allows *Hincent* to scale as discussed in section 4.9.

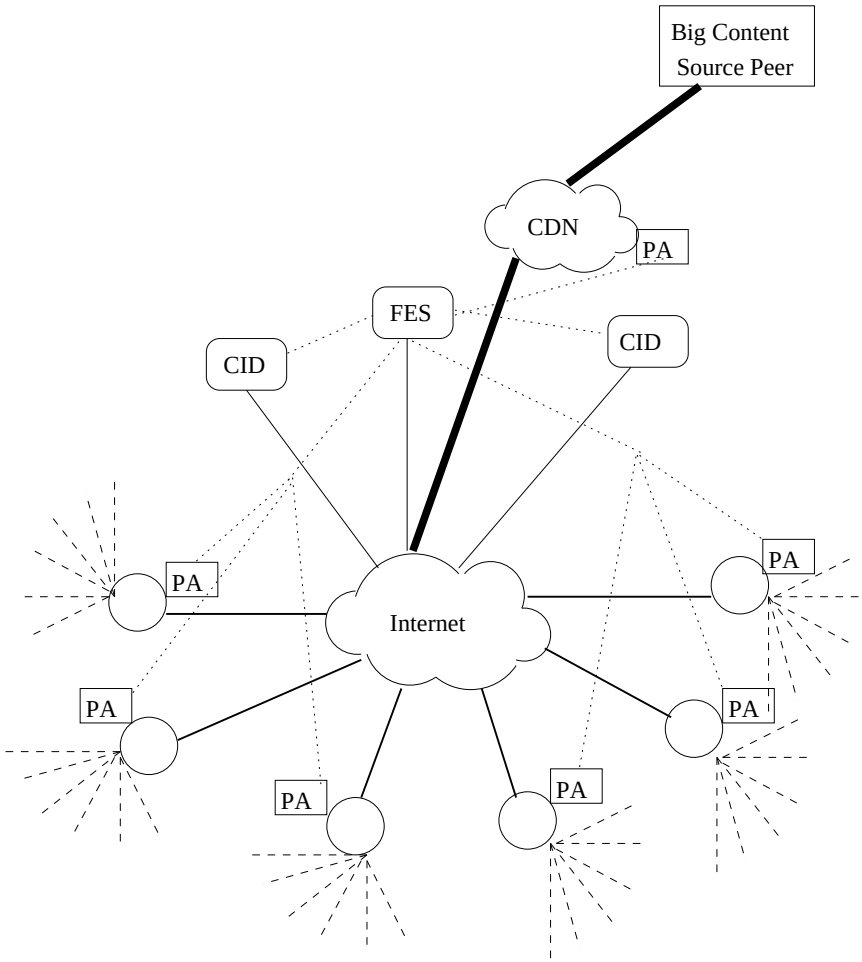

Figure 4.4: The *Hincent* Architecture

4.7.5 Small Content Lifetime (Hop Count) Policy

The peers in the *Hincent* have a strong incentive to store and share the contents they download. As every upload can result in credit which can translate to monetary rewards or high download rate. A node can also inform the content index manager (CIM) that it does not want to serve a specific content. Besides, a peer which has big enough buffer can store early arriving streams to create a 3D tele-Imersive [38] view along with other streams which arrive late. In a scenario where a significant number of peers have limited buffer, *Hincent* can follow a *small content lifetime* policy. In this policy, the CIM uses a content hop count field in its content index database (CID) along with locality information. Here a content source with the lowest hop count is selected to serve the requesting peers. The CIM first tries to find such content in the local CID. If the content with the desired hop count cannot be found in the local CID, it is searched in the master *tblSelectedSource* table as shown in section 4.9.1. If such content with the desired hop count cannot be found, the default *highest rate to price ratio* policy discussed in section 4.4.1 is used.

4.7.6 Private Group Policy

This *Hincent* policy allows content to be shared within a specific group of peers which can be social or organizational groups. Each private group can form its own CIM with any of the above server selection policies. This can enable *Hincent* to deploy Facebook like applications such as the Diaspora [87, 88]. A distributed network of CIMs can also be formed where CIMs exchange public content information based on privacy settings in an adhoc or hierarchical manner. Any peer can then subscribe to different CIMs for different contents forming a distributed content networking.

So far we have been discussing the two major components of *Hincent* which deal with the prioritized rate allocation and resource pricing. After presenting mechanisms of how the uplink and downlink rates for each peer and the corresponding bandwidth prices are calculated by the CIM and PA we have also discussed how the CIM uses these metrics to select a content source for a requesting peer. We next present an efficient content index management scheme which the CIM uses to select the best content source for a requesting peer.

4.8 Content Index Management

In *Hincent*, some peers or content providers provide content by registering their content information at the content index manager (CIM). Other peers request the CIM for a specific content. In this section we show how such contents are registered, requested and their source selected.

4.8.1 Content Index Database

The registered content information is stored at the content index database (CID) which is part of the CIM system as shown in figure 4.2. The CID consists of the *tblPeer*, *tblContent*, *tblSelectedSource*, *tblRequested-Content* tables as shown in figure 4.5. The *tblPeerContent* table is used to link the *tblPeer* and *tblContent* in a many-to-many relationship.

Peer Table

The *tblPeer* contains the fields described in table 4.2. Initial content providers need to fill in all the fields of this table. The *peerInfo* contains real content provider information such as telephone number, address and/or credit card number. Such confirmed information holds each content provider accountable for the nature of the content provided. The *peerInfo* field is also used by the content providers to charge peers for none-free contents. Once a peer receives a content, the CID registers the peer as having the content unless

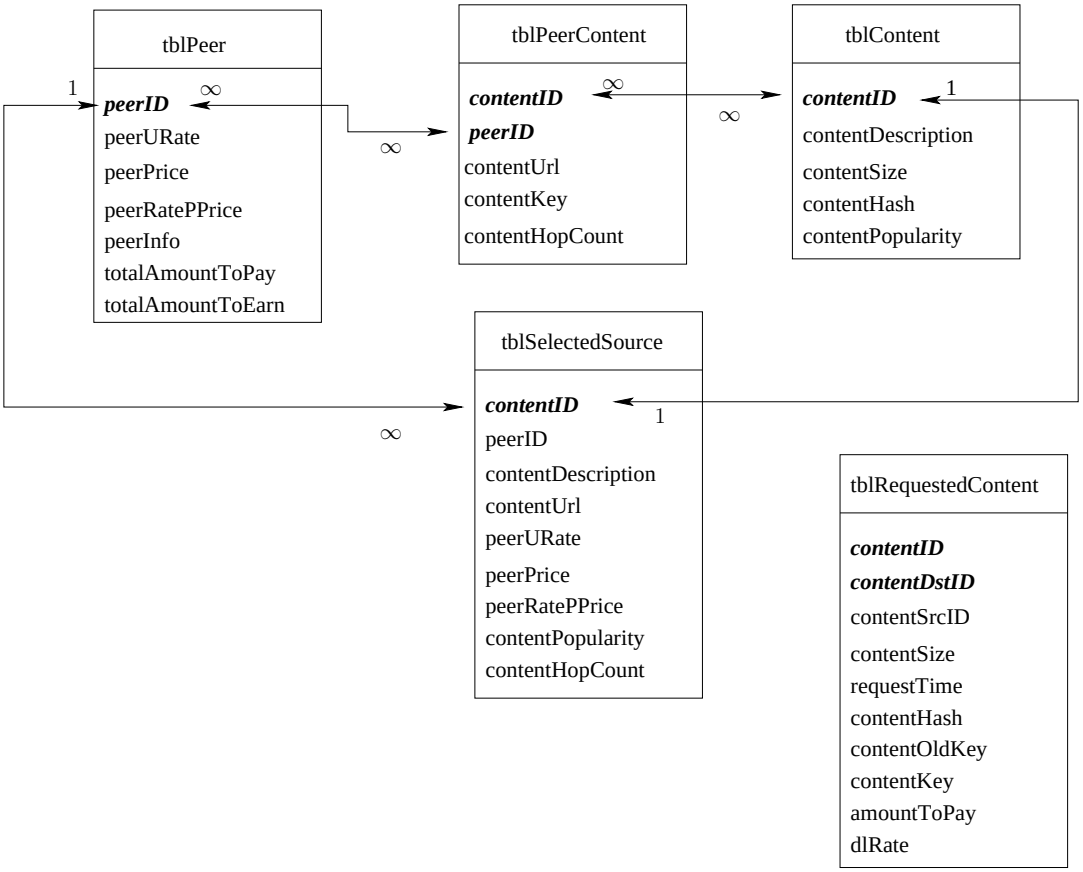

Figure 4.5: The CID Architecture

the peer indicates that it does not want to serve the content. The peers which are not the original sources of the content do not have to provide their *peerInfo* unless they want to receive monetary value of the credit they earn. The *peerID* field is the primary key of the *tblPeer*. It is preferred to be the IP address of the peer. The peers have incentives to provide their correct IP addresses. This is because if a peer gives a wrong IP address, it can not get a content as a source sends its content to an IP address it obtains from the CIM.

Table 4.2: Peer Table Fields

Field name	Description
peerID	Unique peer identifier
peerURate	Current base uplink rate, $R_u(t)$ of a peer calculated using equation 4.5
peerPrice	Per unit uplink cost of a peer node
peerRatePPrice	Peer rate per price calculated using equation 4.14
peerInfo	Real content provider information
totalAmountToPay	Total monetary amount a peer needs to pay for downloading a content
totalAmountToEarn	Total monetary amount a peer earns for uploading a content

If the peer is just joining the CIM, its uplink rate, *peerURate*, is the total uplink capacity it uses to earn credit from other peers to which it uploads content. The peer has an incentive to dedicate more uplink capacity, as more uplink capacity can bring the peer more credit (monetary values). After the initial calculation by the CIM using equation 4.5, *peerURate* is updated by each peer using equation 4.10. To minimize the computation load of the CID, the *peerURate* can also be entirely calculated by the peers in a distributed manner and sent to the CIM every control interval τ. If the peers send their download rates to the CID and if there is enough server processing (computation) capacity at the CIM, all rate computations given by equations 4.5 and 4.10 can also be done by the CIM servers in a centralized manner. In this chapter we use the approach where initial simple rate computation is done by the CIM servers and the more detailed rate update computation is done by the peers. The peers then send the update to the CIM servers.

The *peerPrice* in our study is per packet cost where one packet in this study is 1000 Bytes. The *peerPrice* is initially set to be the unit content cost plus basic initial user defined link cost. The content cost is zero for a free content scenario and the initial link cost in our study is determined by the CIM system. After the initial cost, *peerPrice* is calculated adaptively by the CID servers using equation 4.7 for each link.

The default content source selection policy we use in this study is the *Highest rate to Price Ratio Policy* discussed in section 4.4.1. To implement this policy the *tblPeer* maintains the peer rate per price *peerRatePPrice* field.

The *amountToPay* and *amountToEarn* fields are updated by the *tblRequestedContent* table. A peer gets an additional amount in dollars for each content it serves and pays a certain amount for each content it downloads.

Content Information Table

The second table the CID keeps is the content information table which we call *tblContent* in this chapter. This table contains the fields shown in table 4.3.

Table 4.3: Content Information Table Fields

Field name	Description
contentID	Uniquely identifies content chunk or stream in the CIM
contentDescription	A textual description of the content
contentSize	Size of the content in KB
contentHash	To check for content integrity
contentPopularity	The number of times a content with *contentID* is requested

The *contentSize* is used by the CIM to charge the peer which receives the content. The CID uses this content size to obtain the per content amount a peer has to pay. The *contentHash* is used by the content receiving peer to check for content integrity. Every time a content is selected by a peer, the popularity of the content increases.

Peer-Content Linking Table

This *tblPeerContent* links the *tblPeer* with the *tblContent* in a many-to-many relationship. To achieve this, *tblPeerContent* consists of the primary keys *peerID* and *contentID* of *tblPeer* and *tblContent* tables respectively. The table also contains the peer specific fields, *contentUrl*, *contentKey* and *contentHopCount*. The current location of the content in a peer with *peerID* is *contentUrl*. The source peer encrypts its content with the symmetric key *contentKey*, K as shown in figure 4.6. After a peer receives a content from another peer or from a the original content server, it requests the CID (*tblRequestedContent*) for the key to decrypt the content. The *contentHopCount* is set to 1 if the peer is the original content source. Every other peer which receives the content increments the value of the field by 1. This field along with locality information for instance helps estimate the streaming content age since its initial distribution.

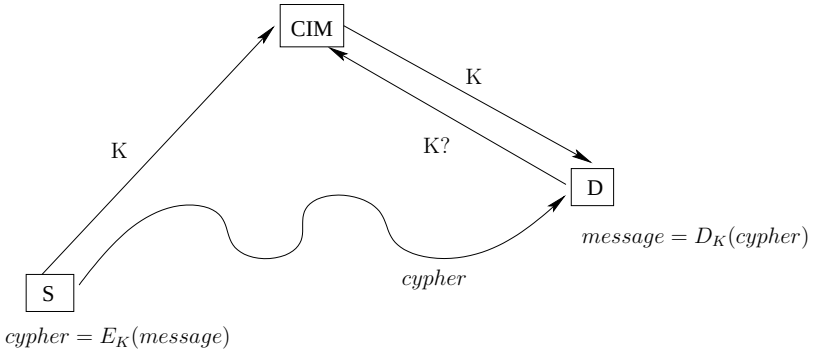

Figure 4.6: Encrypted Content Transfer Using CIM

Selected Source Table

From all the original content servers and peers which have a specific content, a source for a requested content is selected based on the content source selection policy discussed in section 4.4 above. For each content source selection policy, a table called *tblSelectedSource* is produced by a query from the *tblPeer*, *tblPeerContent* and *tblContent* tables. For the *Highest Rate to Price Ratio Policy* (HRPR) used in this chapter, the *tblSelectedSource* has the fields, *contentID*, *peerID*, *contentDescription*, *contentUrl*, *peerURate*, *peerPrice peerRatePPrice* and *contentPopularity*. This table can be sorted in descending order of popularity to put the most popular contents at the top even though every content can be looked up in constant time.

Requested Content table

The requested content table, *tblRequestedContent*, consists of the fields, *contentID*, *contentSize*, *contentSrcID*, *contentDstID*, *requestTime*, *contenOldtKey*, *contentKey*, *amountToPay* and *dlRate*. The *contentID* and *contentSize* fields correspond to the the requested content. The *contentSrcID* field is the *peerID* of the peer or server which is selected to serve the content. The *contentDstID* field is the *peerID* of the content requesting peer. The field, *requesTime* is the time when a request for the specific content was made. The *contenOldtKey* field is a symmetric key with which the content was encrypted and by which the content receiver will decrypt the content. Once a peer with *contentDstID* requests for this key to decrypt the content it downloaded, its *amountToPay* value is set to the product of the *contentSize* and the per packet price, *peerPrice*, of the source link. The *totalAmountToPay* of the peer with *contentDstID* and the *totalAmountToEarn* of the peer or server with *contentSrcID* that serves the content each increase by *amountToPay*. The *contentKey* field is a new symmetric key generated by the CID for the content downloaded by the peer. The content requesting peer uses this key to encrypt the content when selected by the CIM to serve the content. Once the *contenOldtKey* is successfully received by the peer which requested the content, and after other tables of *contentSrcID* and *contentDstID* are updated, the record entry of these fields in *tblRequestedContent* is deleted. The *dlRate* field is set to the minimum of the downlink (to the destination) and uplink (from the source) rates of the requested content.

In the next section we discuss how the CID tables scale with the growth in the number of content and peer record entities.

4.9 Scaling User and Transaction Management

In this section we discuss how *Hincent* scales to an increase in the number of users and with the multiple variations in the request arrival and completion patterns. The CID tables can be scaled with increasing number of peers and contents by using multiple data center like servers along with appropriate hash functions. If the number of servers available for the *tblPeer* table is S_p, $sigBits(peerID)$ gives the integral value corresponding to the most significant bits of the peerID field. How many significant bits of the peerIDs we take depends on how many content entries we have. Taking fewer significant bits for instance means we need fewer servers (smaller S_p) as more peerIDs can be mapped to a single server. A record for *peerID* goes to *tblPeer* located at server $sigBits(peerID)\ mod\ S_p$. Here the servers are identified by positive integral values and mod is the modulo operation. A record for *contentID* of *peerID* goes to *tblContent* located at server $sigBits(peerID)\ mod\ S_p$. This ensures that the content and peer information are located in the same server for easier local look-up.

Such hashing by *sigBits(peerID)* helps that content information of peers whose IP addresses have the same domain go to the same server. In this case if a peer in one index server is selected by the CID as a source of a content to another peer in the same index server, then the content source selection strategy becomes local. Such local content source selection mechanism can help peers achieve low download latency as the content can be served from another peer in their local network.

When the request arrival and completion vary so much, the PA needs to recompute the rate given by equation 4.10 multiple times. Furthermore the PA needs to update the rate values at the respective CIM. Since each CIM obtains temporary rates using equation 4.5, the PA does not have to send every update to the CIM. The PA can send updates every user-defined control intervals.

Equation 4.5 used by the CIM only needs one subtraction (addition) and one division per new flow request arrival or departure to obtain a temporary uplink rate for each peer. It also needs one multiplication and one division to obtain the temporary price given by equation 4.14. Since the process is adaptive, some CIM rate and price updates can as well be skipped as they can be updated by the rate the PA send for each of their links. The CIMs using equation 4.5 also do not need to obtain the temporary downlink rates and prices. The downlink rates and prices can be sent by the content requesting peer. In these cases the CIM only needs to check if the selected source has enough remaining upload link capacity to satisfy a minimum rate requirement $M_{d,u}^{j}$ of the request j.

4.9.1 Database Partition and Aggregate

Assigning *tblPeer* and *tblContent* tables to different index servers based on the peer ID (IP) essentially partitions the CID into multiple local databases. Each local database matches content source and destination peers located in the same network domain and local area. We call such content matching a *local content source selection strategy*.

If content cannot be found in a local network or if peers in other local networks can provide a higher upload rate and lower price, then the source selected to serve a content can be from a different network domain, different area or even different country. Such a content source selection strategy where a content source can be chosen from a different network domain (area) is called *global content source selection*.

To achieve global content source selection, the CID needs to know a source with the highest upload rate and lowest per packet price for the requested content. The CIM achieves this by using a map/reduce [89] like framework as shown in figure 4.7. For the content source selection strategy we use in this chapter, each local CID database's *tblPeerContent* is sorted in descending orders by *contentPopularity* and then *peerRatePPrice* for each content. So here we have each *tblPeerContent* information *mapped* to many local index servers (many CIDs).

For each content, a record with the highest *peerRatePPrice* among all the *tblContent* tables in each local CID database is selected *(reduced)* into the *tblSelectedSource* table and placed in another index server. This is like the reduction phase in the map/reduce framework where the *maximum of* is the reduce function. Each CID continuously sorts the *tblPeerContent* table by the *peerRatePPrice* field for each content with the changes in the upload rates and prices of the corresponding peer.

If the value of *peerRatePPrice* field in a *tblPeer* table changes, first, each *contentID* content of the *peerID* peer in the *tblPeerContent* is sorted in descending order of *peerRatePPrice*. Then for each content of *peerID* in the *tblSelectedSource* table, if the highest *peerRatePPrice* of *peerID* is higher than the *peerRatePPrice* of the corresponding content in *tblSelectedSource*, then the values of the *peerID* and *peerRatePPrice* fields in the *tblSelectedSource* table are replaced with the corresponding values in the *tblPeer*. The *tblSelectedSource* table is sorted by *peerID*. Hence all contents of the *peerID* field are located once the first content of *peerID* is found. Such a procedure of constantly updating the *tblSelectedSource* table ensures that the table always consists of the list of contents given by the peers with the highest upload rate and lowest price (highest rate to price ratio).

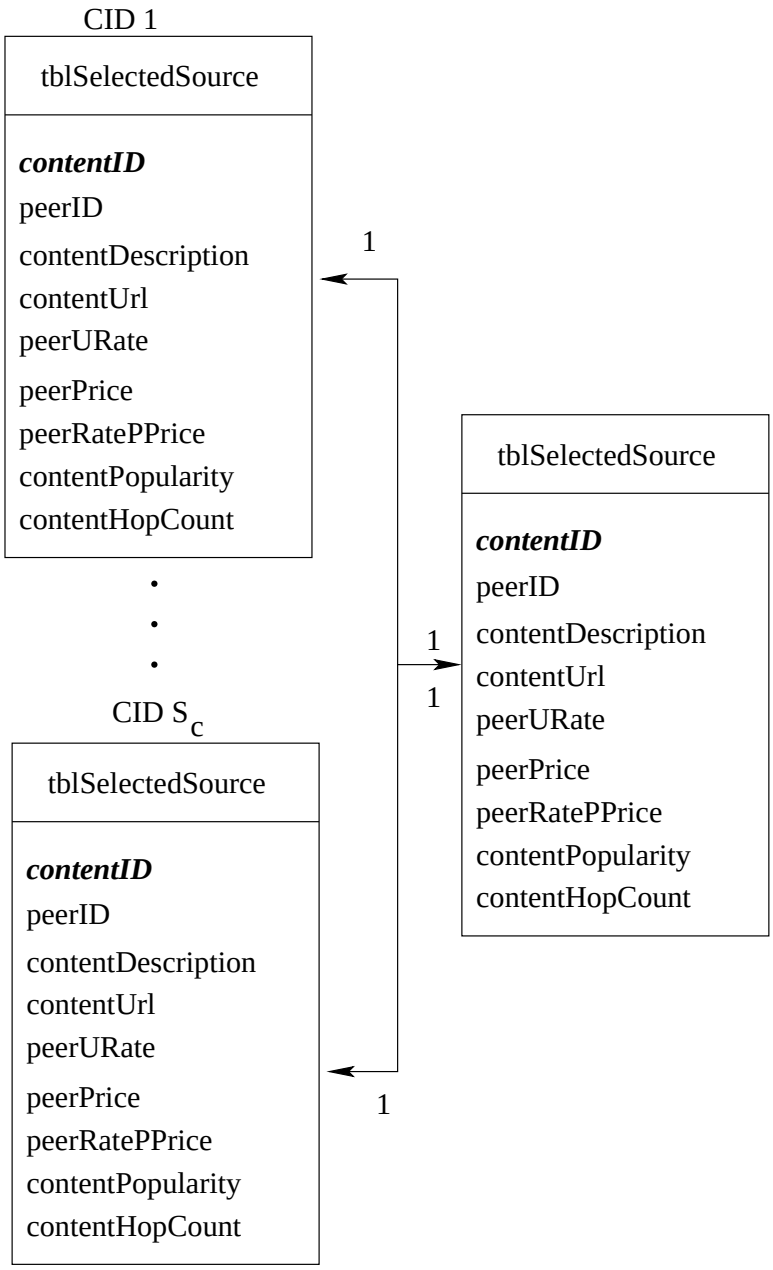

Figure 4.7: The CID Partition and Aggregation

4.9.2 CID Complexity Analysis

The CID operation of adding new peers to the *tblPeer* is of constant order $\mathrm{O}(1)$ as the *tblPeer* does not have to be sorted out. Each content in the *tblPeerContent* has to be sorted out by *peerRatePPrice*. Hence adding a new content entry to the *tblPeerContent* table has a complexity of order $\mathrm{O}(log(N_c))$ where N_c is the number of peers which have the same content.

Whenever a peer gets a content that it requested, then (1) the uplink rate of the content source decreases according equation 4.5 and (2) the peer which gets the content becomes one of the content sources. These two operations require two $\mathrm{O}(log(N_c))$ operations for each content. As the *tblPeerContent* in each of the CID partition is sorted, updating the *tblSelectedSource* for each of its contents is of constant order.

When *tblRequestedContent* is updated upon a successful download of a content by a peer, the corresponding values of *peerRatePPrice* and *totalAmountToEarn* of a source peer or server and the *totalAmountToPay* of a receiving peer are updated in constant time by using the matching *peerID* field.

4.10 *Hincent* with Scarce Backbone Bandwidth

The backbone links in the Internet which the nodes use and which are represented by the "Internet" node in figure 4.1 are not usually congested as can also be seen from [9]. Each user of the *Hincent* mechanism can also have bandwidth service level agreement from the operators which guarantee the desired capacity. Under this scenario the only bottleneck links are the last links to and from the *Hincent* peer nodes. Hence, the sources of the desired contents can set their congestion window sizes (*cwnd*) to the product of their uplink rate value calculated using equation 4.11 and their round trip time (RTT).

If the bottleneck link is somewhere in the Internet which is described as "Internet" node in figure 4.1, then the destination of flow i sets its receive window size as given by equation 4.2. And the source of flow i obtains its maximum congestion window size w_M^i using equation 4.4.

A node can detect whether or not the bottleneck is in the link other than the last links to and from the source and destination peers using different ways. For instance, if a packet loss is observed for flow i after the rate is enforced using equation 4.3, then the TCP source of flow i can assume the bottleneck link is other than the last links to/from the source and destination nodes. The PAs of the receiving end can also count the number of received packets (bytes) per unit time to obtain the actual download rate per content. Similarly, the PA of the content source can also estimate its uplink rate of a specific content by counting the number of successfully acknowledged packets (bytes) per unit time. The PA of the source and destination of the content then report this rate to the CIM per specific content. The CIM then replaces the *peerURate*

of the content in *tblSelectedSource* with the minimum of these two values. The source and destination peer also update their rate calculations using equations 4.8, 4.9 and 4.10 to re-allocate unused capacities to other requests.

The *tblRequestedContent* of the CID has the *requestTime* and *contentSize* fields. When a peer receives a content, it requests the CID for the decryption key. The CID of the CIM can then compare the $actualTime = currentTime - requestTime$ against $promisedTime = contentSize \,/\, dlRate$, where $currentTime$ is the time when the request for the decryption key arrives at the CIM and $dlRate$ is the minimum of the uplink rate and downlink rate of the requested content. If the $actualTime > promisedTime + toleranceVal$, then the CIM via its CM concludes that the source is not uploading at the rate it suggested. Here $toleranceVal$ is a user-defined tolerance value. In cases where the requested content is a video stream, the source of the content can stream its frames by scheduling them at $\frac{1}{R_u^i}$ apart where R_u^i is given by equation 4.11.

4.11 *Hincent* Using Surrogate Servers

In the *Hincent* deployment scenario presented in the previous sections, each peer uses a personal web (content) server similar to the Diaspora social network [87, 88]. The personal web server can be hosted at a home server, at a friend server or at an ISP. An extension of *Hincent* can also be implemented in big content distribution services such as Google (YouTube) or other overlay (private) networks such as [85] using cloud/cloudlet surrogate servers geographically distributed in a wide area as described in figure 4.8. The servers are equipped with OpenFlow vSwitches (switches/routers). The links of the network shown in figure 4.8 can be dedicated tunnels or overlay links over the Internet. If the links are overlay, their capacity can be estimated using bandwidth estimators. We next discussed how *Hincent* distributes content using such surrogate servers for the peers uploading and requesting for content using OpenFlow vSwitchs [90, 45].

As shown in figure 4.8, peer clients upload and get contents from their nearest surrogate servers using the following steps.

Content storage steps:

- A peer client which wants to share its content with other peers or which wants to store its data in some distributed servers, sends its request to a light weight FES (frontend server) associated with the controller shown in figure 4.8. The FES can also be associated with each surrogate server.

- The FES hashes the request and forwards it to the closest server using (first significant bits of) IP address matching. This ensures that a request is handled by surrogate server in the locality of the requesting peer.

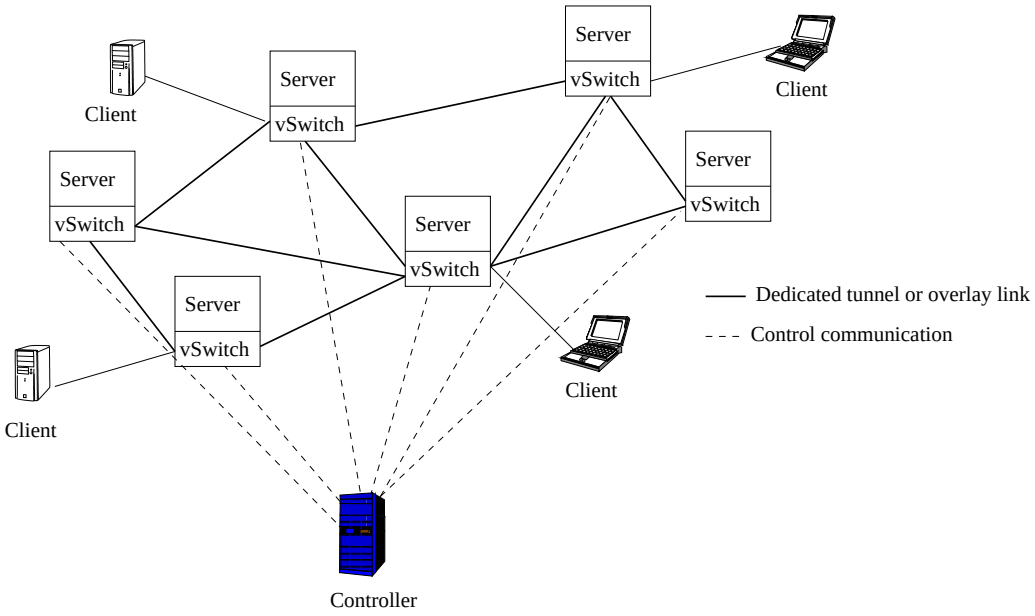

Figure 4.8: *Hincent* with Surrogate Servers

- The FES also forwards the content information to a modified *tblContent* which resides in the CID (content index database) of the controller. The modified *tblContent* includes the ID of the selected (by FES) surrogate server where the content initially resides in addition to the other fields described in table 4.3.

- The controller informs all surrogate servers of the new content. The surrogate server which is initially selected to host the content can also share the contentID information with the other servers similar to the way link state information is shared.

- Each surrogate server adds a new record to its *tblServContent* table which has the fields *contentID*, *contentPopularity*, *sourceID* and *servURate*. The *contentID* and *sourceID* values are obtained from the controller or from the surrogate server which is selected to initially host the content. The value of *servURate* along with the path to the server with the content is obtained using a max/min routing algorithm described in [39] and [40]. The link metric of the network of surrogate servers described in figure 4.8 is a cross-layer (routing and congestion control) rate metric obtained using the schemes described in [40]. As discussed in [40], this rate metric can be obtained using vSwitch (OpenFlow) per flow packet counts (statefull) or using surrogate server assistance (stateless).

- Each surrogate server updates its *tblServContent* table sorting each contentID entry in decreasing order of *servURate* value every time route computation to other servers is done. The servers also sort the content entries in decreasing order of popularity. When a peer request for content is made, a selected

server gets the content from another server with the highest *servURate* (for the requesting peer). Here *servURate* is the bottleneck update rate from a source surrogate server to the destination surrogate server.

Content retrieval steps:

- A peer client requests for a content by contacting the FES. The FES seamlessly hashes the client ID and forwards its request to a surrogate server which is the nearest (closest IP address for instance) to the requesting client.
- If the surrogate server has the content, it directly transmits it to the requesting client. Otherwise, the surrogate server looks up its *tblServContent* table for the best (highest *servURate*) other server with the requested content. It then starts a QCP [40] session (can also be TCP) to the selected (highest *servURate*) server and gets the requested content for the requesting peer.
- The server which downloaded the content on the behalf of the peer client stores (caches) the content and informs the controller CID that it also has the content. The controller CID then informs other servers that the server also has the content.
- This surrogate server which obtained the content from another server also transfers the content to the requesting peer. The connection between the peer client and its surrogate server can also be QCP if the peer has a dedicated tunnel connecting it with its server. Otherwise it can be a TCP connection.

We have performed detailed experimental analysis to evaluate the performance of *Hincent* as shown in the next sections.

4.12 Evaluation

In this section we evaluate the performance of *Hincent* and all its components using simulation. We implemented *Hincent* in the NS2 simulation package. We also implemented the CID of *Hincent* using Apache SQL server [91]. After discussing the simulation setup, we present detailed trace-based packet level simulation experiments. We then show how *Hincent* content management scales using Apache SQL implementation experiments.

4.12.1 Simulation Setup

We use a simulation topology similar to the one given in figure 4.1. For the simulation the upload and download capacities of the links to and from the peers is 15Mbps. The link capacity to and from the CDN is

$n_{peers} \times 15\ Mbps$, where n_{peers} is the number of peers. The propagation delay between the peers is taken from 4 hour PlanetLab traces [92]. The average CDN bandwidth price taken from the Amazon CloudFront [86] is $avg\_cdnPrice$ = \$0.176 per GB of traffic. The initial peer bandwidth price is $avg\_cdnPrice/(2.0 \times n_{peers})$. This price adaptively increases as the peer rate decreases with more demands based on equation 4.7. We run different sets of experiments as shown in the following sections.

4.12.2 Pure CDN Vs *Hincent*-Based Schemes

Figures 4.9 and 4.10 show how the *Hincent*-based scheme scales with the growing number of content requesting peers when compared with the pure CDN-based approach. This result is consistent with detailed study [33] which shows that the hybrid CDN-P2P can significantly reduce the cost of content distribution bandwidth.

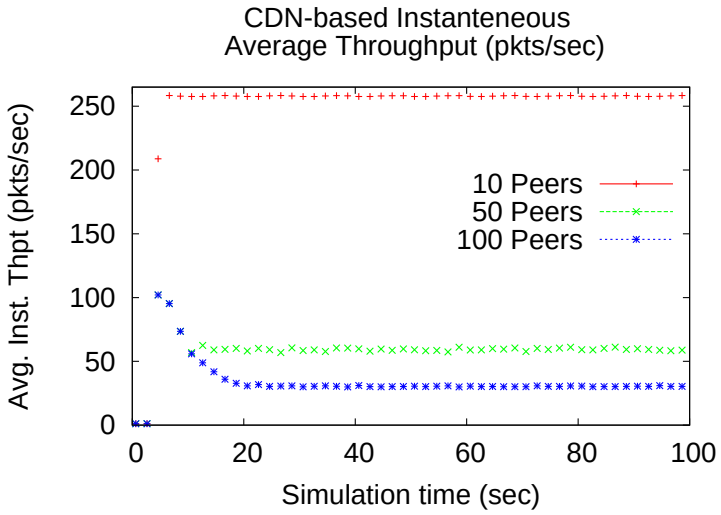

Figure 4.9: Pure CDN based Approach

4.12.3 Other P2P schemes Vs *Hincent*

We have also compared the performance of *Hincent* against other hybrid P2P and CDN schemes in terms of average chunk completion time (ACCT). Previous hybrid P2P and CDN schemes such as the Dandelion [31], PACE [36] use TCP as their transport protocol. So we show how these schemes using TCP compare against *Hincent* by fixing the content source selection mechanism to be the same (based on *Hincent*) for both.

For this experiment we use 8 files with content $i\,, (1 \leq i \leq 8)$ having file size $500i$ KB and chunk size i is $50i$ KB. Inter-content chunk request time is 0.5 seconds. Contents are requested at the same time. Each file (content) is divided into equal chunks. Content popularity is 5 for each of the contents. For the TCP-based

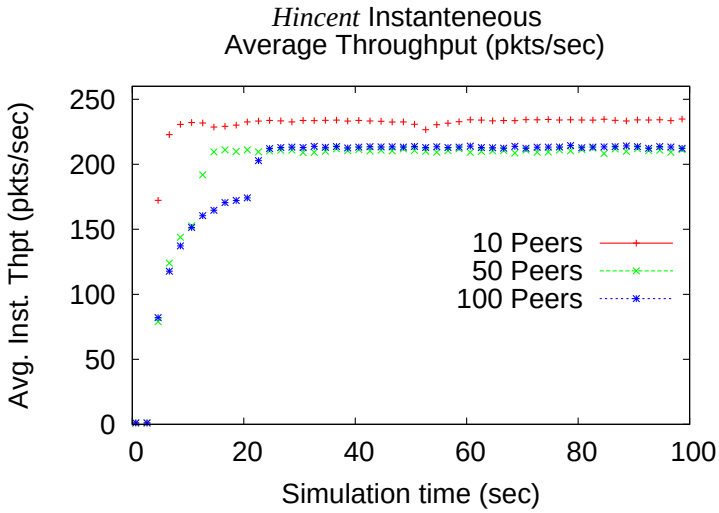

Figure 4.10: *Hincent*-based Approach

and the *Hincent* approaches content destination and source are the same. For these experiments we set the minimum flow rate to 0.0 and all chunks have the same priority levels.

Figure 4.11 shows that the ACCT and average maximum CCT (Max CCT) are much smaller in *Hincent* than the TCP-based approaches (PACE, Dandelion). The Max CCT is the content (file) completion time as a file download is complete after its latest chunk is downloaded.

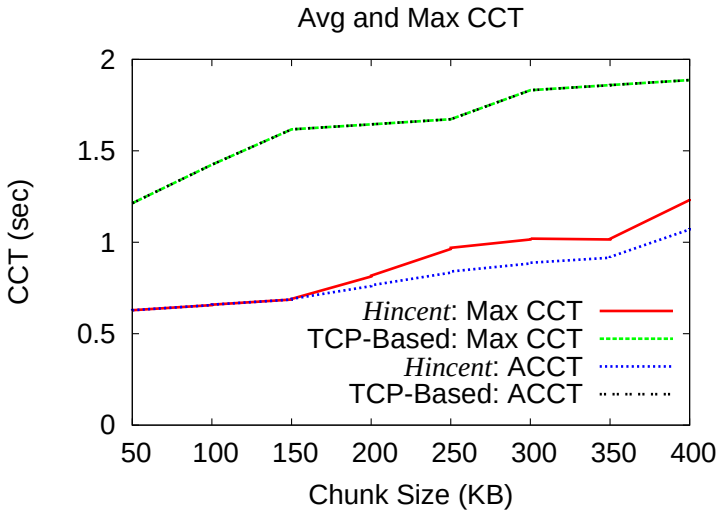

Figure 4.11: Avg and Max CCT of *Hincent* Vs TCP-Based Approaches

4.12.4 3D Streaming Result

For the 3D streaming experiments, we use a setup which emulates [85] with 6 streams. Each stream demands a minimum of 1Mbps capacity. Each stream i, $1 \leq i \leq 6$ has a priority weight of $1/i$. We used a content lifetime of 2.5 seconds for the streaming. So if a stream at a peer is older than 2.5 seconds, the CIM does not register the peer as having the content.

Figure 4.12 demonstrates the priority and minimum rate mechanisms of *Hincent*. As shown in the figure, stream 1 which has the highest priority weight gets highest throughput. The throughput of the other streams follows their priority weights.

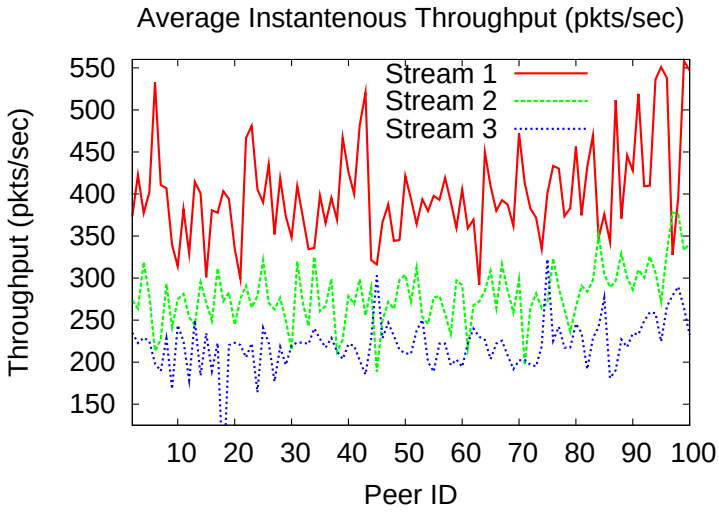

Figure 4.12: Avg Instantaneous Throughput Per Peer for Streams 1, 2 and 3

Figure 4.13 also shows how the instantaneous throughput of the different streams evolve with time. All these plots show how efficiently *Hincent* enforces the priority based rate allocations. For readability and in the interest of space, plots with streams 4, 5 and 6 are omitted.

4.12.5 More Trace-Based Experiments

We have also conducted experiments based on the trace results presented in [76] for the content size distribution, [78] for the content popularity distribution and [77] for distribution of the flow arrival process. Since we could not obtain the raw trace data, we constructed the trace values (data points) from the plots given in these papers. We next present the trace extraction methodologies we used.

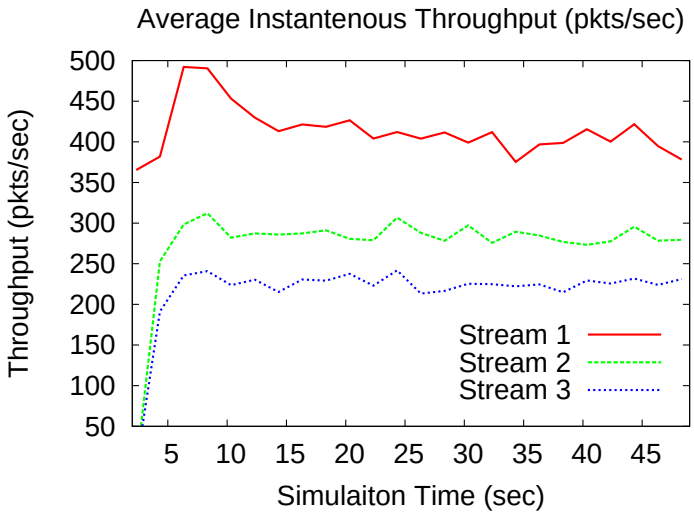

Figure 4.13: Avg Instantaneous Throughput Over Time for Streams 1, 2 and 3

Extracting file size distribution

Based on the nature of the file size trace plot of US-Campus given in figure 4 of [76], we constructed piece-wise linear functions given by equation 4.15.

$$FS = \begin{cases} u(1,5), & cdf <= 0.17, \\ \frac{10-5}{0.18-0.17}(u(0,1)-0.17)+5.0, & 0.17 < cdf \leq 0.18, \\ \frac{200-10}{0.204-0.18}(u(0,1)-0.18)+10, & 0.18 < cdf \leq 0.204, \\ \frac{1000-200}{0.25-0.204}(u(0,1)-0.204)+200, & 0.204 < cdf \leq 0.25, \\ \frac{30000-1000}{0.96-0.25}(u(0,1)-0.25)+1000, & 0.25 < cdf \leq 0.96, \\ \frac{70000-30000}{0.99-0.96}(u(0,1)-0.96)+30000, & 0.96 < cdf \leq 0.99, \\ \frac{100000-70000}{1.0-0.99}(u(0,1)-0.99)+70000, & 0.99 < cdf \leq 1.0. \end{cases} \quad (4.15)$$

In equation 4.15, the function $u(a,b)$ generates a uniform random number between a and b and cdf is the CDF of the file size trace plot. In the simulation, we first generate a uniform random number between 0 and 1. We then obtain the file size (FS) value as a function of the generate value using equation 4.15.

Extracting content popularity distribution

A Gamma distribution curve with a shape parameter of $\tilde{k} = 0.372$ and a scale parameter of $\theta = 23910$ is fitted to Youtube video content popularity distribution traces in figure 7 of [78]. The content popularity distribution in the paper which refers to the number of views of videos considers about $N_V = 1.6 \times 10^5$ videos. We normalized the scale parameter θ of the distribution by the number N_V of distinct videos so as to use it with simulation studies involving a different number of videos. The normalization steps are as

follows.

With n_v as the total number of distinct (unique) video flows to be simulated, and p_v the average popularity of the videos, $n_v p_v$ is the total number of videos to be simulated. With a simulation time of t_s seconds and average video request arrival rate of λ_s flows per second, we have

$$n_v = \frac{t_s \lambda_s}{p_v}. \tag{4.16}$$

To obtain p_v, we normalize the number N_V of traced videos by the mean $\tilde{k}\theta$ of the Gamma popularity distribution as

$$\frac{n_v}{p_v} = \frac{N_V}{\tilde{k}\theta}. \tag{4.17}$$

Combining equations 4.16 and 4.17, we get the popularity value as

$$p_v = \sqrt{\frac{\tilde{k}\theta t_s \lambda_s}{N_V}}. \tag{4.18}$$

Using equation 4.18 in equation 4.16 we also obtain the number of distinct videos in the simulation.

Flow arrival distribution

We used the distribution of the number of flow arrivals per second given in [77] for our simulation. The paper fits a Poisson distributed curve to the trace and hence we used such a distribution for our flow arrivals. The number of YouTube servers (servers with unique IP addresses) used in the experiment was 2138. To scale our simulation we considered arrival rates to 1 and 10 servers. The experiment can simply be run for all servers with more powerful machines.

More Trace Experimental Results

To compare the performance of pure *Hincent* based approach against other TCP based approaches (PACE, Dandelion), we considered the best case scenario for the TCP based approaches. This scenario uses the *Hincent* content selection mechanism (see section **??**). So using this same server selection mechanism we compared the performance of the TCP-based approaches with our pure *Hincent* based approach. As can be seen from figures 4.14, 4.15 and 4.16, the pure *Hincent* approach gives lower file completion time when compared with TCP-based *Hincent* approach. For all experiments in this section, each YouTube file is divided into 50 chunks. So bigger file sizes have bigger chunk sizes. The YouTube video files we consider in this analysis are not live videos. Hence we use a content age of 15.5 seconds. This implies that videos which

were first requested less than 15.5 seconds ago can still be requested. For all experiments of one YouTube server, the Intel i5 Core machine we used allowed us to run the simulation for 120 seconds. For the 10 YouTube servers experiments, we used a simulation time of 30 seconds.

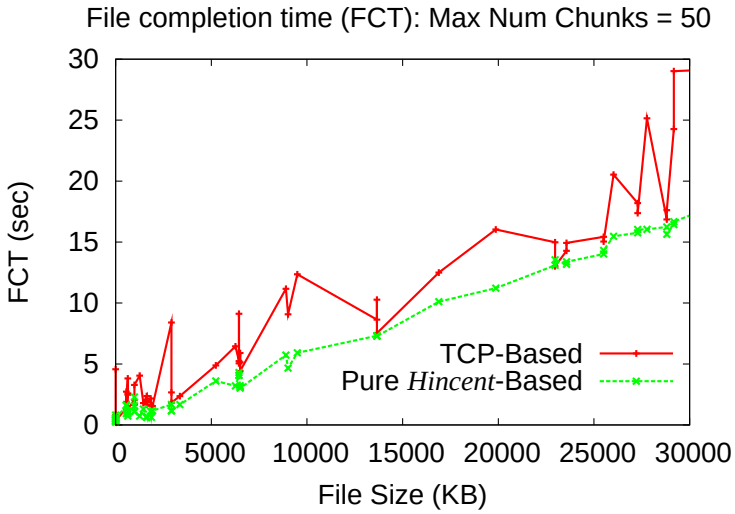

Figure 4.14: File completion time with 1 YouTube server

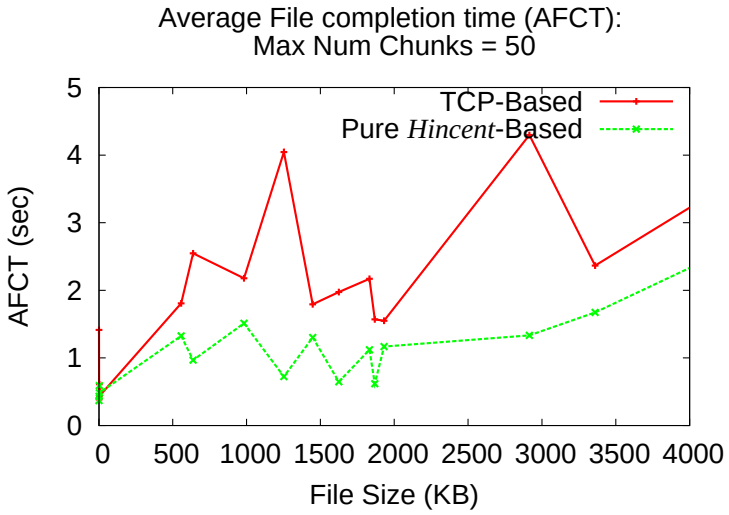

Figure 4.15: Average file completion time (AFCT) with 1 YouTube server (small files)

Figure 4.15 shows the average file completion time (AFCT) of files less than 4000KB in size while figure 4.14 shows FCT of all files. As can be seen from figure 4.16, with more YouTube servers, the number of simulated peers requesting for content increases. This in turn increases the number of peers with a content and hence decreasing the file download time (AFCT). This is one of the noble gains of peer to peer systems as more peers means more bandwidth.

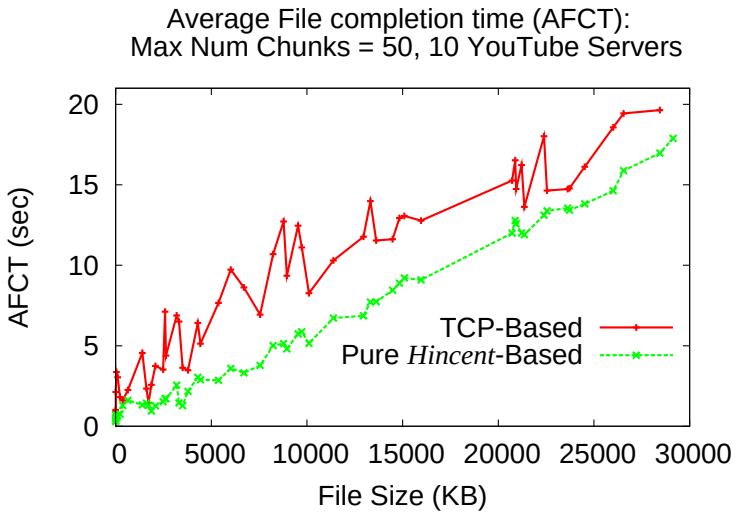

Figure 4.16: Average file completion time (AFCT) with 10 YouTube server

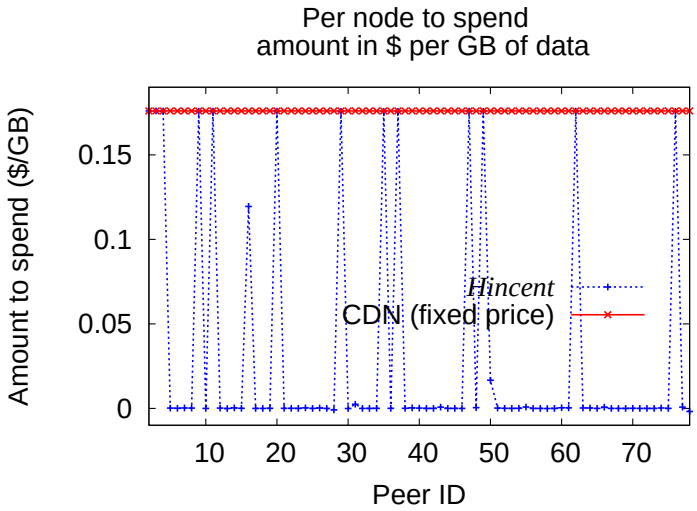

Figure 4.17: Net amount to pay in dollars per GB of downloaded content with 1 YouTube server

Figures 4.17, 4.18 and 4.19 show that overwhelming majority of the peers do not have to spend money to download GB of data as the credit amount they earn balances out with the amount they pay. For each peer, the amount to spend in these plots is calculated as the total amount of money a peer earns minus the total amount a peer has to pay per GB of content.

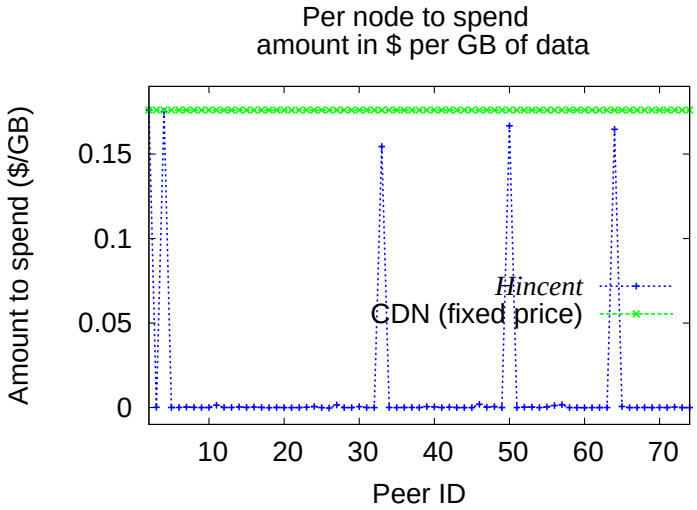

Figure 4.18: Net amount to pay in dollars per GB of downloaded content with 10 YouTube servers (First few peers)

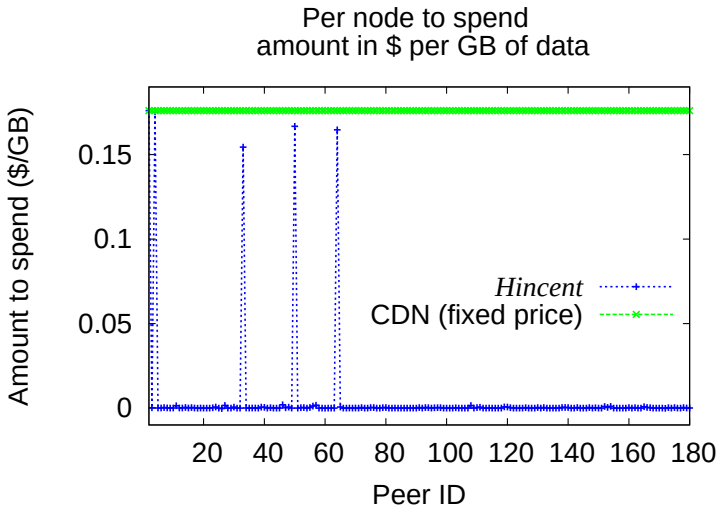

Figure 4.19: Net amount to pay in dollars per GB of downloaded content with 10 YouTube servers (All peers)

Comparing figures 4.17 and 4.19, it can be seen that more YouTube servers in the experiment means more participating peers. The more peers have the contents the less other peers have to download the content

from the CDN servers. This saves peers more money as can be seen from the plots. In all cases, the amount peers pay for bandwidth to download a content is less than the fixed CDN bandwidth amount charged by AmazonCLoudFront. For the experiments with only one YouTube server, the simulation generates fewer peers to download the content. As the number of peers which have the content is smaller, more peers download contents from the CDN servers paying more money as can be seen in figure 4.17. The amount which peers pay to directly download a content from the CDN servers can be subsidized (paid for) by the content providers as such peers are serving as seeders for the content provider.

4.12.6 CID Implementation Experiments

We have also implemented the basic features of *Hincent* in an Apache SQL server using PHP script. We implemented all the tables of the CID in an Ubuntu virtual machine using a quad four processor and a 1GB RAM. We generated *tblSelectedSource* table using a SELECT query from the tables *tblPeer*, *tblContent*, *tblPeerContent* as discussed in section 4.8.1 above. The tables are linked in a many-to-many relationship.

To see the performance gain of using the *tblSelectedSource* table over generating the contents requested by peers on the fly from the three tables, we have conducted experiments using and not using the *tblSelected-Source* table. We used one million records in each table for this experiment. As can be seen from figures 4.20 and 4.21 preparing the *tblSelectedSource* table as its source tables are updated results in significant gain in query time. Here, query time is the time from when a query for a specific record is made to when the reply is displayed from the SQL server. In these experiments we first generated uniform random content index records with the given contentIDs to request from the SQL server. The content with the ID of *cont396224* was the first content requested. Such initial request of a record resulted in a higher query time perhaps because the SQL server took time to upload parts of the table into memory. Figure 4.21 also shows that the query time increases with the increase in the record ID. This is because the tables are roughly sorted by requestIDs as the none-numeric parts of the contentID and peerID values are the same while both fields have text data types.

The SQL query we made from the *tblSelectedSource* for the content with contentID of *cont396224* is as follows.

```
SELECT *
FROM ‘tblSelectedSource‘
WHERE contentID = ’cont396224’
LIMIT 0 , 1
```

And the following is the query we made from the three tables.

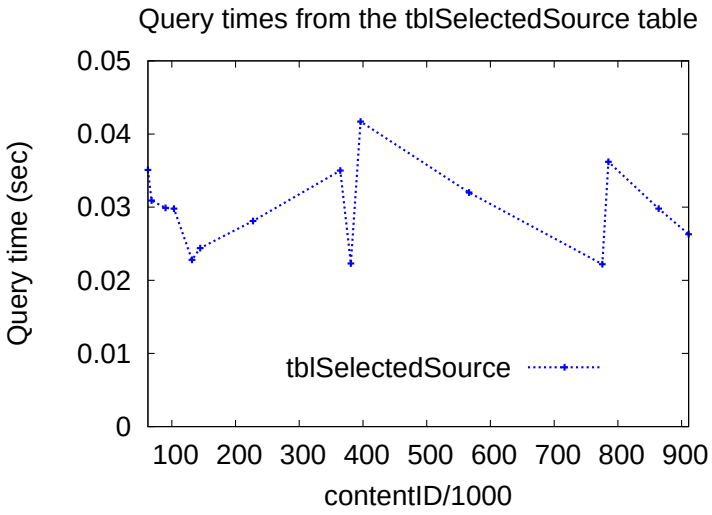

Figure 4.20: Query time using the tblSelectedSource table

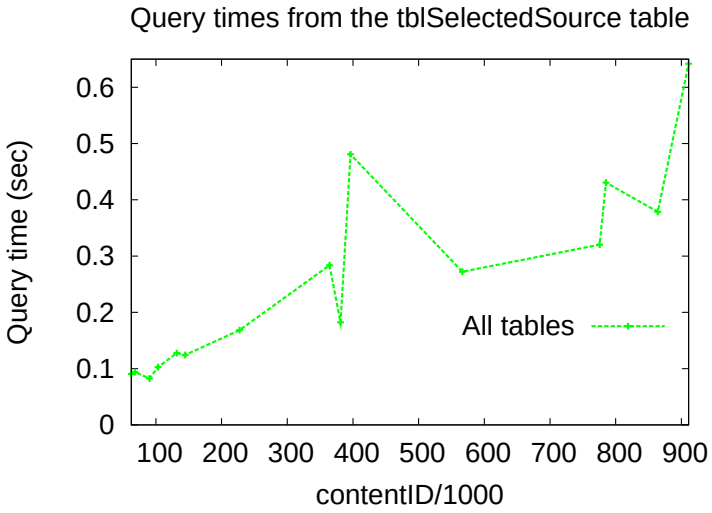

Figure 4.21: Query time using SQL JOIN from all tables

```
SELECT tblPeerContent.contentID, tblPeerContent.peerID,
  tblPeerContent.contentUrl, tblPeerContent.contentKey,
  tblContent.contentDesc, tblContent.contentPopularity,
  tblPeer.peerURate, tblPeer.peerUPrice, tblPeer.ratePerPrice
FROM tblPeerContent
INNER JOIN tblPeer ON tblPeerContent.peerID = tblPeer.peerID
INNER JOIN tblContent
  ON tblPeerContent.contentID = tblContent.contentID
WHERE tblPeerContent.contentID = 'cont396224'
LIMIT 0 , 1
```

We next conducted an experiment to know how long it takes for a query such as requesting the *contentKey* by a peer from the CID of the CIM. The propagation delay from the requesting peer virtual machine to the virtual machine with the SQL server is about 1ms. The times it takes for such query is shown in figure 4.22. There is a spike on the record of *cont132913* which is the first record requested by the peer in the experiment. Such a spike disappears with the other requested records as perhaps the SQL server caches the session and keeps the *tblRequestedContent* table loaded in memory.

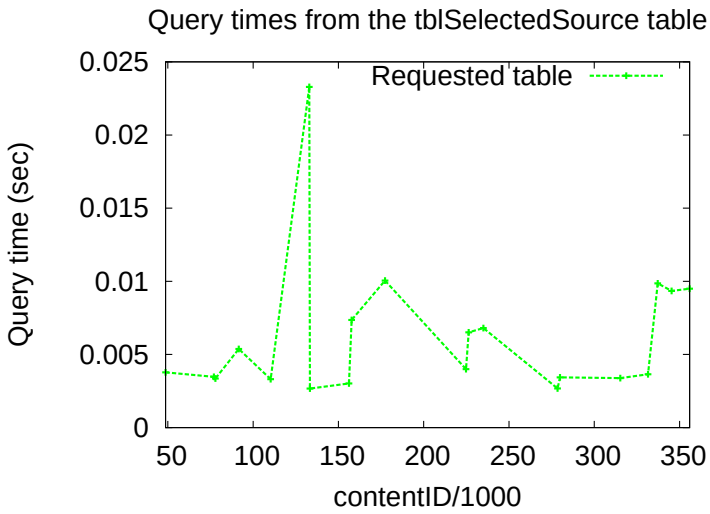

Figure 4.22: Query time from peers to the CIM

The query we used for the experiments in figure 4.22 is as follows.

```
SELECT * FROM tblRequestedContent
WHERE contentID = 'cont$value'
```

```
LIMIT 0,1
```

These above query time figures are intended to show that the time it takes to resolve a certain query is not high even using a computer (server) with very limited hardward such as an Ubuntu virtual machine.

4.13 Related Work

Over time, Peer-to-peer (P2P) content distribution has evolved to incorporate incentives in order to prevent freeloading. The BitTorrent [93, 94] uses a rate based tit-for-tat mechanism where users can achieve higher download rate from peers to which they are uploading. In this case a peer which is not downloading a content is not incentivized to upload a content. In *Hincent* all peers are incentivized to continue uploading as every upload increases their credit maintained by the *Hincent* CIM. Reputation based schemes such as [95] help peers find another peer with the highest reputation score to download content from. Such a reputation scheme does not provide an accurate evaluation mechanism to choose a peer to serve a content. For instance a peer which is uploading many files without downloading a file can have a high reputation score. If such a peer does not have as much available upload capacity as another peer which is downloading files, peers will select it anyways because it has a high reputation score.

In the KARMA [96] scheme, every peer has a set of managers which form banks which coordinate credit transfer with other peers. In this scheme there is no guarantee of integrity of the global currency when the majority of the managers are malicious. In *Hincent* a central CIM which cannot be manipulated by peers offers real monetary rewards to all peers which upload contents. PACE [36] uses bandwidth pricing to help uploading peers earn credit. However PACE does not give a fair-exchange of content for payment as the content demand at a peer is estimated as a total requested download rate at remote buy clients. Such demand used to obtain a bandwidth price is not peer specific. Dandelion [31] is based on a centralized online currency bank mechanism to incentivize peers. However Dandelion uses a fixed pricing mechanism that peers are not awarded according to the upload bandwidth they offer to upload contents. Peers do not decrease their price to attract more customers when they have high upload rate and vice versa. PRIME [97] is a mesh-based P2P streaming. Even though it tries to balance the average outgoing rate of a source peer with the average incoming rate of a content receiving peer, it does not use an efficient rate allocation and enforcement mechanism like *Hincent* to achieve a max/min allocation. It uses a TCP friendly rate control protocol (TFRC) [98] which inherits the TCP problems of not quickly utilizing available link capacities. In PRIME each peer tries to maintain many parents that can collectively serve as content providers using a mesh-based overlay construction which can potentially incur significant overhead. Unlike *Hincent*, PRIME does not give

an efficient mechanism to help peers select a content source with high throughput and minimum bandwidth cost. This is because a new peer selects a random subset of peers to be its content parents. A reliable client accounting system of a commercial hybrid content-distribution network (Akamai) is also presented in [99] to detect and mitigate a variety of attacks by malicious peers. This mechanism improves the NetSession which is a peer-assisted content delivery network (CDN) operated by Akamai. In *Hincent* peers do have any incentive to act maliciously. This is because peers get monetary incentives (credit) for uploading content and all transactions are co-ordinated by a scalable centralized *Hincent* CIM. If an *Hincent* peer acts maliciously, it only wastes its bandwidth and suffers monetary losses.

A hybrid CDN-P2P system for live video streaming called LiveSky is presented in [100]. The paper gives a trace based study of extensive LiveSky deployment in China. However the work only gives approximate guideline for peer selection. For instance the paper assume that the total upload bandwidth of clients in level k of the P2P tree is always larger than the download bandwidth requirement of clients in level k+1. It also only considers aggregate measures (i.e., population and time averages) to model the end-user properties. On the other hand *Hincent* does not make such assumptions and uses accurate rate and price based incentives to select content sources to serve a content. This gives peers a reliable incentive to cooperate without a malice. LiveSky also limits peer selection to a local network while *Hincent* does not make that ristriction unless local content source selection stragy is used or the local peers have the best upload rate and lowest prices. A study in [23] shows that redirecting every client to the CDN server with least latency does not suffice to optimize client latencies. The authors of this paper proposed a system called *WhyHigh* to optimize Google CDN perfomance. WhyHigh measures client latencies across all nodes in the CDN and correlates measurements to identify the prefixes affected by inflated latencies. *Hincent* by design chooses peers or CDN servers which offer the highest throughput and lowest price and does not require complex inefficient systems such as WhyHigh to select content sources.

NetTube, a P2P assisted content delivering framework that explores the clustering in social networks for short video sharing is proposed in [101]. Like NetTube, *Hincent* allows users to share their contents while keeping it in their own servers. Unlike *Hincent*, NetTube selects a content source based on social groups and not based on throughput and bandwidth price. SocialTube, which is peer-assisted video sharing system that explores social relationship, interest similarity, and physical location between peers in online social networks (OSNs) is proposed in [102]. SocialTube uses a social network (SN)-based P2P overlay construction algorithm. Unlike *Hincent*, SocialTube does not select content sources based on a high upload bandwidth and low cost. This can result in SocialTube unnecessarily delaying streaming and other content transfer when other peers not in the same social group with high upload capacity exist.

Besides, unlike *Hincent*, all the above schemes do not help peers determine an accurate rate at which they can download content from other peers. They do not give a mechanism to prioritize content transfers which is an important component of 3D [38] and other streaming applications. Unlike *Hincent* they also do not provide an efficient max/min rate allocation mechanism.

4.14 Summary

In this chapter we proposed the design of *Hincent*, an efficient cross-layer content routing and congestion control framework. Unlike existing content distribution approaches, *Hincent* relies on an accurate and fair incentive mechanism which allows prioritized max/min rate allocations and enforcements. *Hincent* is a flexible scheme which allows multiple server selection strategies. Unlike previous work we have presented a noble content index management scheme for *Hincent*. It allows distributed peers to have full control of their contents and to securely share them with others. We have also presented an extension of *Hincent* using surrogate servers with OpenFlow vSwitches to help peers exchange contents faster than using existing schemes.

We have implemented *Hincent* in the NS2 simulation package. We evaluated the performance of *Hincent* using rigorous trace based flow and packet level simulation experiments. The experiments demonstrate the *Hincent* design goals which result in lower content transfer time than existing schemes. We have also implemented *Hincent* content index management with Apache SQL server using PHP in Ubuntu virtual machines. The implementation experiments show that *Hincent* can easily scale to millions of content index records.

Chapter 5

Conclusion and Future Work

In this thesis, we first formulate a common *congestion control and routing problem* shared by major distributed systems such as the *Internet* (general networks), *datacenter* and hybrid P2P networks. The problem formulation specifies many essential requirements and challenges (constraints). We show how existing systems fail to sufficiently address the problem and the challenges/constraints associated with it. We then present our unified *cross-layer* routing and congestion control framework to address the challenges of the major distributed systems.

Our framework derives an optimal *rate* metric which is used as a joint congestion control and routing metric. It is used to set the *content transfer rates* (flow rates) and also as a link weight metric to find optimal *routes*. Using the rate metric, we present architectural details for each distributed system to achieve, among other things, lower content transfer delay and higher and smoother throughput when compared with well known existing schemes.

We implemented our cross-layer protocols for the three major distributed systems in the NS2 simulation. Detailed trace-based and packet level simulation experiments validate our design goals of achieving lower content transfer times and higher and smoother throughput than well known existing architectures. We have implemented the content index management (CIM) component of our hybrid P2P architecture with Apache SQL server using PHP in Ubuntu virtual machines. The CIM can scale to millions of content index records as shown in the implementation experiments.

Large scale testbed implementation and testing of our cross-layer schemes *(QCP, SCDA, Hincent)* is left for future work. We also plan to conduct detailed simulation and testbed implementation experiments of using the SCDA software components (RM, RA) in complex general networks. Similarly, the *Hincent* feature using surrogate servers also merits detailed simulation and proof-of-concept testbed implementation to fully test and validate it.

References

[1] J. T. Moy, *OSPF: Anatomy of an Internet Routing Protocol*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[2] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet inter-domain traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, Aug. 2010.

[3] C. Labovitz, "Internet traffic evolution 2007-2011," http://www.monkey.org/ labovit/papers/gpf_2011.pdf.

[4] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky, "Overclocking the Yahoo!: CDN for faster web page loads," in I*MC '11*. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2068816.2068869 pp. 569–584.

[5] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879175 pp. 267–280.

[6] F. Qian, A. Gerber, and *et.al*, "TCP revisited: a fresh look at TCP in the wild," in I*MC '09*. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644903 pp. 76–89.

[7] A. Rao, A. Legout, and *et.al*, "Network characteristics of video streaming traffic," in C*oNEXT '11*. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2079296.2079321 pp. 25:1–25:12.

[8] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 59–62, 2006.

[9] Internet2, "Internet2 Network: GlobalNOC RealTimeAtlas," ttp://atlas.grnoc.iu.edu/I2.tml, 2012.

[10] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14 –23, oct.-dec. 2009.

[11] C.-H. Chi, D. Gasevic, and W.-J. van den Heuvel, Eds., *16th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2012, Beijing, China, September 10-14, 2012*. IEEE, 2012.

[12] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*. New York, NY, USA: ACM, 1988, pp. 314–329.

[13] T. R. Henderson, E. Sahouria, S. McCanne, R. H. Katz, and Y. H. Katz, "On Improving The Fairness Of TCP Congestion Avoidance," in *IEEE Globecomm*, 1997, pp. 539–544.

[14] T. V. Lakshman and U. Madhow, "The performance of tcp/ip for networks with high bandwidth-delay products and random loss," *IEEE/ACM Trans. Netw.*, vol. 5, no. 3, pp. 336–350, 1997.

[15] J. Rojas-Mora, T. Jiménez, and E. Altman, "Simulating flow level bandwidth sharing with pareto distributed file sizes," in VALUETOOLS '11. ICST, Brussels, Belgium, Belgium: ICST, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2151688.2151718 pp. 265–273.

[16] S. Floyd, "HighSpeed TCP for Large Congestion Windows." United States: RFC Editor, 2003.

[17] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 89–102, 2002.

[18] A. Akella, S. Chawla, A. Kannan, and S. Seshan, "Scaling properties of the internet graph," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, ser. PODC '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/872035.872087 pp. 337–346.

[19] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: responsive yet stable traffic engineering," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 253–264, August 2005. [Online]. Available: http://doi.acm.org/10.1145/1090191.1080122

[20] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture." United States: RFC Editor, 2001.

[21] K. Kar, M. Kodialam, T. V. Lakshman, and S. Member, "Minimum interference routing of bandwidth guaranteed tunnels with mpls traffic engineering application," *IEEE Journal on Selected Areas in Communications*, p. 2579, 2000.

[22] J. Wang, Y. Yang, L. Xiao, and K. Nahrstedt, "Edge-based traffic engineering for ospf networks," *Comput. Netw.*, vol. 48, pp. 605–625, July 2005. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2004.11.008

[23] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao, "Moving beyond end-to-end path information to optimize cdn performance," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644917 pp. 190–201.

[24] S. Triukose, Z. Wen, and M. Rabinovich, "Measuring a commercial content delivery network," in *Proceedings of the 20th international conference on World wide web*, ser. WWW '11, 2011, pp. 467–476.

[25] 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 *Proceedings of the ACM SIGCOMM '09*, 2009, pp. 51–62.

[26] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX NSDI'10*, 2010, pp. 19–19.

[27] V. Jacobson, "Congestion avoidance and control," in *Symposium proceedings on Communications architectures and protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988. [Online]. Available: http://doi.acm.org/10.1145/52324.52356 pp. 314–329.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945450

[29] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972 pp. 1–10.

[30] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 63–74, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851192

[31] M. Sirivianos, X. Yang, and S. Jarecki, "Robust and efficient incentives for cooperative content distribution," *IEEE/ACM Trans. Netw.*, vol. 17, pp. 1766–1779, Dec. 2009.

[32] P. Wendell and M. J. Freedman, "Going viral: flash crowds in an open cdn," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 549–558.

[33] C. Huang, A. Wang, J. Li, and K. W. Ross, "Understanding hybrid cdn-p2p: why limelight needs its own red swoosh," ser. NOSSDAV '08. New York, NY, USA: ACM, 2008, pp. 75–80.

[34] M. Marcon, B. Viswanath, M. Cha, and K. P. Gummadi, "Sharing social content from home: a measurement-driven feasibility study," ser. NOSSDAV '11. ACM, 2011, pp. 45–50.

[35] C. Aperjis, M. J. Freedman, and R. Johari, "Peer-assisted content distribution with prices," ser. ACM CoNEXT '08. New York, NY, USA: ACM, 2008, pp. 17:1–17:12.

[36] C. Aperjis, R. Johari, and M. J. Freedman, "Bilateral and multilateral exchanges for peer-assisted content distribution," *IEEE/ACM Trans. Netw.*, vol. 19, no. 5, pp. 1290–1303, Oct. 2011.

[37] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki, "Dandelion: cooperative content distribution with robust incentives," in *2007 USENIX Annual Technical Conference*, ser. ATC'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 12:1–12:14.

[38] Z. Yang, W. Wu, K. Nahrstedt, G. Kurillo, and R. Bajcsy, "Enabling multi-party 3d tele-immersive environments with viewcast," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 6, no. 2, pp. 7:1–7:30, Mar. 2010.

[39] D. Fesehaye, I. Gupta, and K. Nahrstedt, "A Cross-layer Routing and Congestion Control for Distributed Systems," University of Illinois at Urbana-Champaign (UIUC), TECHNICAL REPORT, Nov. 2008. [Online]. Available: https://www.ideals.illinois.edu/handle/2142/11503

[40] D. Fesehaye and K. Nahrstedt, "Finishing Flows Faster with A Quick congestion Control Protocol (QCP)," University of Illinois at Urbana-Champaign (UIUC), TECHNICAL REPORT, 01 2013. [Online]. Available: https://www.ideals.illinois.edu/handle/2142/35905

[41] D. Fesehaye and K. Nahrstedt, "Hincent: Quick Content Distribution With Priorities and High Incentives," in *IEEE Consumer Communications and Networking Conference (CCNC 2013)*, Las Vegas, USA, jan 2013.

[42] D. Fesehaye and K. Nahrstedt, "SCDA: SLA-aware Cloud Datacenter Architecture for Efficient Content Storage and Retrieval," in *8th IEEE International Conference on Networking, Architecture, and Storage (NAS 2013)*, Xian, China, jul 2013.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[44] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks (OpenFlow White Paper)," March 2008. [Online]. Available: http://www.openflow.org/

[45] B. Pfaff and et. al., "Openflow switch specification v1.3.0," https://www.opennetworking.org/, April 2012.

[46] D. Fesehaye, "A Fast Congestion control Protocol (FCP) for Networks (the Internet)," in *ITRE*, 2006, pp. 131–135.

[47] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, pp. 344–357, June 1993. [Online]. Available: http://dx.doi.org/10.1109/90.234856

[48] L. Kleinrock and R. R. Muntz, "Processor sharing queueing models of mixed scheduling disciplines for time shared system," *J. ACM*, vol. 19, pp. 464–482, July 1972. [Online]. Available: http://doi.acm.org/10.1145/321707.321717

[49] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, Aug. 1989. [Online]. Available: http://doi.acm.org/10.1145/75247.75248

[50] H. Balakrishnan, N. Dukkipati, N. McKeown, and C. Tomlin, "Stability analysis of explicit congestion control protocols," *Communications Letters, IEEE*, vol. 11, no. 10, pp. 823 –825, october 2007.

[51] Z. Lu and S. Zhang, "Stability analysis of xcp congestion control systems," in *Wireless Communications and Networking Conference, 2009. WCNC 2009. IEEE*, april 2009, pp. 1 –6.

[52] S. Higginbotham, "Google's next OpenFlow challenge: taking SDNs to the consumer," http://gigaom.com/cloud/how-google-is-using-openflow-to-lower-its-network-costs/, apr 2012.

[53] S. Higginbotham, "How Google is using OpenFlow to lower its network costs," http://gigaom.com/cloud/how-google-is-using-openflow-to-lower-its-network-costs/, apr 2012.

[54] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my asic!" ser. HotSDN '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342447 pp. 25–30.

[55] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using cpu as a traffic co-processing unit in commodity switches," ser. HotSDN '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342448 pp. 31–36.

[56] PluribusNetworks, "F64 Series Server-Switch with Netvisor OS," http://www.pluribusnetworks.com/solutions/, 2012.

[57] M. Scharf and H. Strotbek, "Performance evaluation of quick-start TCP with a Linux kernel implementation," in I*FIP NETWORKING 2008*. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792514.1792592 pp. 703–714.

[58] J. C. Mogul, J. Tourrilhes, and *et.al*, "DevoFlow: cost-effective flow management for high performance enterprise networks," ser. Hotnets '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868448 pp. 1:1–1:6.

[59] Z. Cai, A. L. Cox, and T. S. N. Eugene, "Maestro: A System for Scalable OpenFlow Control," Technical Report: Rice University, 2010.

[60] T. Koponen, M. Casado, and *et.al*, "Onix: a distributed control platform for large-scale production networks," ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968 pp. 1–6.

[61] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter," HotNets, 2009.

[62] Nicira, "Its Time to Virtualize the Network," http://nicira.com/en/network-virtualization-platform, 2012.

[63] S. McCanne and S. Floyd, "ns-2," http://www.isi.edu/nsnam/ns/.

[64] A. Greenberg, J. R. Hamilton, and *et.al*, "VL2: a scalable and flexible data center network," *Commun. ACM*, vol. 54, no. 3, pp. 95–104, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1897852.1897877

[65] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya, "End-to-end performance implications of links with errors." United States: RFC Editor, 2001.

[66] E. Kohler, M. Handley, and S. Floyd, "Designing dccp: congestion control without reliability," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 27–38, 2006.

[67] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP vegas: new techniques for congestion detection and avoidance," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 24–35, Oct. 1994. [Online]. Available: http://doi.acm.org/10.1145/190809.190317

[68] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, "TCP Westwood: end-to-end congestion control for wired/wireless networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 467–479, Sep. 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1016590112381

[69] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[70] N. Dukkipati, N. McKeown, and A. G. Fraser, "RCP-AC: Congestion Control to Make Flows Complete Quickly in Any Environment," in *INFOCOM*, 2006.

[71] P. Pillay-Esnault., P. Moyer, J. Doyle, E. Ertekin, and M. Lundberg, "Ospfv3 as a provider edge to customer edge (pe-ce) routing protocol," United States, 2012.

[72] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of the ACM SIGCOMM '09*, 2009, pp. 39–50.

[73] R. T. Kaushik and M. Bhandarkar, "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster," in *Proceedings of the 2010 international conference on Power aware computing and systems*, ser. HotPower'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924920.1924927 pp. 1–9.

[74] D. Fesehaye, R. Malik, and K. Nahrstedt, "Efficient Distributed File System (EDFS)," University of Illinois at Urbana-Champaign (UIUC), TECHNICAL REPORT SLIDES, 07 2010. [Online]. Available: https://www.ideals.illinois.edu/handle/2142/16563

[75] R. T. Kaushik and K. Nahrstedt, "T: a data-centric cooling energy costs reduction approach for big data analytics cloud," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389067 pp. 52:1–52:11.

[76] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao, "Dissecting video server selection strategies in the youtube cdn," ser. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 248–257.

[77] T. Mori, R. Kawahara, H. Hasegawa, and S. Shimogawa, "Characterizing traffic flows originating from large-scale video sharing services," ser. TMA'10. Springer-Verlag, 2010, pp. 17–31.

[78] X. Cheng, C. Dale, and J. Liu, "Statistics and social network of youtube videos," in *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, june 2008, pp. 229 –238.

[79] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2079296.2079304 pp. 8:1–8:12.

[80] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967 pp. 63–74.

[81] C. Hopps, "Analysis of an equal-cost multi-path algorithm," United States, 2000.

[82] A. López-Ortiz, "Valiant load balancing, benes networks and resilient backbone design," in *Proceedings of the 4th conference on Combinatorial and algorithmic aspects of networking*, ser. CAAN'07. Berlin, Heidelberg: Springer-Verlag, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1778487.1778490 pp. 2–2.

[83] D. Fesehaye, R. Malik, and K. Nahrstedt, "EDFS: a semi-centralized efficient distributed file system," in *Middleware (Companion)*, 2009, p. 28.

[84] D. Fesehaye, R. Malik, and K. Nahrstedt, "A Scalable Distributed File System for Cloud Computing," University of Illinois at Urbana-Champaign (UIUC), TECHNICAL REPORT, 03 2010. [Online]. Available: https://www.ideals.illinois.edu/handle/2142/15200

[85] W. Wu, A. Arefin, Z. Huang, P. Agarwal, S. Shi, R. Rivas, and K. Nahrstedt, ""i'm the jedi!" - a case study of user experience in 3d tele-immersive gaming," ser. ISM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 220–227.

[86] Amazon, "Amazon cloudfront," http://aws.amazon.com/cloudfront, 2012.

[87] "The diaspora* project," http://diasporaproject.org/.

[88] K. Weise, "On diaspora's social network, you own your data," http://www.businessweek.com/articles/2012-05-10/on-diasporas-social-network-you-own-your-data, 2012.

[89] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[90] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual switching in an era of advanced edges," Sep. 2010. [Online]. Available: http://openvswitch.org/

[91] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou, "Sql server column store indexes," ser. ACM SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1177–1184.

[92] "Planetlab 4hr traces," www.eecs.harvard.edu/syrah/nc/sim/pings.4hr.stamp.gz.

[93] B. Cohen, "Incentives build robustness in bittorrent," Proc. Workshop Econ. Peer-to-Peer Syst, 2003.

[94] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do incentives build robustness in bit torrent," ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007.

[95] M. Gupta, P. Judge, and M. Ammar, "A reputation system for peer-to-peer networks," ser. NOSSDAV '03. New York, NY, USA: ACM, 2003, pp. 144–152.

[96] V. Vishnumurthy, S. Chandrakumar, , and E. G. Sirer., "Karma: A secure economic framework for peer-to-peer resource sharing," Proc. Workshop on Economics of Peer-to-Peer Systems, 2003.

[97] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," *Networking, IEEE/ACM Transactions on*, vol. 17, no. 4, pp. 1052 –1065, aug. 2009.

[98] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "Tcp friendly rate control (tfrc): Protocol specification," United States, 2003.

[99] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon, "Reliable client accounting for p2p-infrastructure hybrids," ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012.

[100] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky," in *Proceedings of the 17th ACM international conference on Multimedia*, ser. MM '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1631272.1631279 pp. 25–34.

[101] X. Cheng and J. Liu, "Nettube: Exploring social networks for peer-to-peer short video sharing," in *INFOCOM*, 2009, pp. 1152–1160.

[102] Z. Li, H. Shen, H. Wang, G. Liu, and J. Li, "Socialtube: P2p-assisted video sharing in online social networks," in *INFOCOM*, 2012, pp. 2886–2890.