Greener Data Exchange in the Cloud: A Coding-Based Optimization for Big Data Processing

Zakia Asad, *Student Member, IEEE*, Mohammad Asad Rehman Chaudhry, *Member, IEEE*, and David Malone

***Abstract*—The rise of the cloud and distributed data-intensive (big data) applications puts pressure on data center networks due to the movement of massive volumes of data. Reducing the volume of communication is pivotal for embracing greener data exchange by efficient utilization of network resources. This paper proposes the use of mixing technique, *spate coding*, working in tandem with software-defined network control as a means of dynamically-controlled reduction in volume of communication. We introduce motivating real-world use-cases, and present a novel *spate coding* algorithm for the data center networks. We also analyze the computational complexity of the general problem of minimizing the volume of communication in a distributed data center application without degrading the rate of information exchange, and provide theoretical limits of such schemes. Moreover, we proceed to bridge the gap between theory and practice by performing a proof-of-concept implementation of the proposed system in a real world data center. We use Hadoop MapReduce, the most widely used big data processing framework, as our target. The experimental results employing two of industry standard benchmarks show the advantage of our proposed system compared to a *vanilla Hadoop implementation*, an *in-network combiner*, and *Combine-N-Code*. The proposed coding-based scheme shows performance improvement in terms of volume of communication (up to 62%), goodput (up to 76%), disk utilization (up to 38%), and the number of bits that can be transmitted per Joule of energy (up to 200%).**

***Index Terms*— Big data, optimization, hadoop, green computing, green communication, cloud computing, spate coding, data center networks, middlebox**

I. Introduction

CICSO global cloud index predicts that by 2017 cloud traffic will represent sixty-nine percent of data center traffic [2]. Furthermore, recent business insights into data center network evolution also forecast an unprecedented growth in data center traffic with 76% of the aggregate traffic not exiting the data center [2]. The increasing migration of applications to the cloud is probably the major driver of this trend: even without a fundamental change in the average communication/computation ratio exhibited by data center workloads, increasing the compute-density-per-server by means of virtualization leads to proportional increase in traffic generated per server. Orthogonally though, to the effects of virtualization, the outset of applications crunching and moving large volumes of data—captured by the market-coined term *big data applications*—is also foreseen to significantly contribute to higher network traffic within data centers [3].

The burden of high volume of communication elemental to big data processing is anchored to network components like switches, and routers. With the advent of energy proportional computations at servers, the energy quota for network components is predicted to soar as high as 50% of the total energy consumption by data centers [4]. Data centers' energy consumption related to the network was reported to be around 15.6 billion KWh in year 2008 [5]. Energy proportional network components play a vital role in reducing a data center's energy consumption. In energy proportional network components, the energy consumption is proportional to the volume of communication. Reduction in volume of communication directly corresponds to energy savings in a data center [4], [6]–[9]. Furthermore, some approaches conserve data center network's energy consumption by adaptively choosing the link rates to match the volume of traffic going through network component. For instance, an energy saving of approximately 4 Watts per link can be achieved by decreasing the link rate from 1 Gbps to 100 Mbps [10]. In such adaptive approaches, decreasing the volume of communication improves the energy profile by favoring lower link rates. Hence, the opportunity to improve the energy efficiency by reducing volume of communication serves as a corner stone in bringing green practices to big data processing.

Even with the provision of the energy-efficient network architecture, an inherent limit in data center network-architectures poses a serious challenge towards embracing big data in an efficient and greener fashion. Specifically, the data center networks still are not ready for digesting petabytes of network data traffic crossing the bisection [11], [12]. Processing data at this speed needs a network with enough bisectional bandwidth to allow every server (node) send data at full speed to every other server (node). This means that network bisection bandwidth is going to cap the rate at which different servers can communicate with each other [13]. Unfortunately,

Manuscript received March 28, 2015; revised July 20, 2015; accepted December 4, 2015. Date of publication January 21, 2016; date of current version May 19, 2016. A preliminary version of this work was presented at the IEEE Syscon 2015 [1].

Z. Asad is with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada (e-mail: z.asad@mail.utoronto.ca).

M. A. R. Chaudhry was with IBM Research, Dublin, Ireland, and the Hamilton Institute, Maynooth University, Maynooth, Ireland (e-mail: masadch@soptimizer.org).

D. Malone is with the Hamilton Institute, Maynooth University, Maynooth, Ireland (e-mail: david.malone@nuim.ie).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2016.2520245

the data center networks are optimized for North-South traffic (connections between servers and clients) rather than East-West traffic (servers communicating with each other within data center) since most of the software architectures in the pre-cloud era were meant for clients accessing the servers within data centers. However, with the rise of cloud and distributed architectures like Hadoop MapReduce [14], Storm [15], and Dryad/DryadLINQ [16], [17] that are implemented on the different nodes across the data centers, applications rely on East-West traffic for most of their computations. In fact, as a defacto standard, data centers use the tree topology assuming that network bisection offers enough bandwidth for the machines at the lower levels in the network topology [12]. However, most of the distributed application big data applications like Hadoop MapReduce exchange tremendous amounts of data over highly oversubscribed links resulting in high packet drop rates [11]. Therefore, unfolding the full potential of big data by greener data centers calls for efficient utilization of network resources, and new non-intrusive frameworks that can seamlessly integrate with existing network architectures. Consequently, such new frameworks are expected to heavily rely on managing network traffic.

The main driving force for the adoption of cloud data centers is rooted in its ability to deliver services and data at a faster rate, resulting in improved application performance as well as higher operational efficiencies. In order to achieve the desired performance, most of the research is focused on scheduling computation, jobs, and resources (e.g., see [18]–[20] and references therein). However, since more than 50% of the job completion-time of many jobs is consumed in the communication phase [3] it is essential to explore the novel means by which the communication time can be reduced. In this context different approaches have been studied for ameliorating the network effects due communication-intensive workloads by managing flows [3], [21], [22], and demand-driven path alterations [23]–[25]. As essential as it is to ensure higher operation efficiency, avenues to bring in greener prospects in data center networks are equally fundamental. In this vein this work focus on presenting an efficient way to manage the network traffic by minimizing its volume which results in improved resource usage and their energy footprints. Our work is complementary to existing approaches.

This paper explores the potential of integrating novel concepts and techniques of *spate coding* into big data processing frameworks. We propose exploiting a fundamental property common to most big data frameworks, i.e., sharing of a physical node among several processes giving rise to side information. This side information is generated as a consequence of availability of a process's output to other processes hosted by the same node without going through the network. The concept of *spate coding* is inspired by index coding problem [26], [27], though these two are different problems with different solution spaces. The core idea in *spate coding* is to mix (encode) packets, while leveraging the side information, with the objective of reducing the overall volume of communication. Although our proposed solution is general in nature, for demonstration purposes we have used the Hadoop MapReduce framework. Apache Hadoop [14], an open-source implementation of the MapReduce framework, constitutes a popular and thus good representative of data-intensive workloads. It has been shown to scale-out to thousands of nodes and is used across various domains (from bio-sciences to commercial analytics). Focusing on the *shuffle* phase of Hadoop, which is known to be the communication-intensive phase of the application [28], we present in this paper our work on improving the volume of communication, goodput, disk utilization, queue size, and the number of bits that can be transmitted per Joule of energy during the shuffle phase.

A. Contributions

We propose a system for optimizing big data processing in a data center, and present motivating use-cases. Our contributions are not only theoretically significant, but we have also performed a proof-of-concept implementation of the proposed system in a real world data center.

Our major theoretical contributions include analyzing the computational complexity of a general scheme that tries to minimize volume of communication in a distributed data center application without degrading the rate of information exchange. We then present theoretical limits of such schemes by providing upper and lower bounds. Moreover, we have proven that in contrast to a frequent practice in many data centers, the network bisection is not an optimal location for middlebox placement for some applications. Furthermore, we have formulated the *spate coding* problem, which helps in reducing volume of communication in big data applications. We have presented an efficient solution for *spate coding*.

Our major practical contributions include a proof-of-concept implementation of the proposed system in a real world data center, and a testbed. We use Hadoop, the most widely used big data processing framework, as our target framework. We have used two of the industry-standard benchmarks—*Terasort*, and *Grep*—for performance evaluation of the proposed system. The experimental results exhibit coding advantage in terms of volume of communication, goodput, disk utilization, queue size, and the number of bits that can be transmitted per Joule of energy. We also present a novel technique for an efficient implementation of mutlicast leveraging software-defined networking (OpenFlow).

B. Related Work

Many studies have been conducted to optimize data transfer during the communication-intensive phase of big data frameworks [3], [29], [30], spanning from high-level shifting of virtual machines to application-level communication reduction mechanisms (Hadoop combiners) to low-level continuous runtime network optimization in coordination with orchestrators. Moreover, traffic-flow-prediction based scheduling to reduce the impact of network transfers, and improve job processing has been used to optimize network traffic [21], [31]. Another approach for traffic optimization is the use of Redundancy Elimination (RE) schemes that identify and remove repeated content from network transfer (e.g., [32]–[34]) for increasing end-to-end application throughput. We want to point out that all these approaches are complementary to our work, and

can be combined with our scheme to further improve performance characteristics. Moreover, in our scheme only the destination nodes perform the decoding, in contrast to the RE schemes where intermediate nodes are required to be involved in the decoding process. Furthermore, our scheme provides an instantaneous encoding and decoding of flows.

In comparison to the related work, which treats data flow as commodity flow (routing), we utilize the novel concept of mixing (coding), and it has been proven that mixing at the network nodes can provide the optimal rate of information exchange in many scenarios where the routing can not [35]. It has been shown that mixing techniques such as index coding can significantly increase the rate of information transfer in a network [27], [36], [37]. Still, index coding [26], [27], [36]–[39] assumes a broadcast channel, an assumption that is far from reality in a data center. We therefore propose *spate coding* to improve the rate of information transfer in a non-broadcast environment, as is the case with a data center network. We want to point out that the solution techniques as well as theoretical results for index coding do not hold for *spate coding*. Moreover, *spate coding* problem subsumes index coding problem.

II. Motivating Use-Cases

Before delving into system specification details, we present two motivating use-cases using real, albeit toy, examples. The examples showcase the use of mixing (coding) in big data applications, and its potential in reducing volume of communication. The first example is focussed on Hadoop MapReduce, whereas the second one is focussed on Storm.

A. Hadoop MapReduce for Electricity Theft Detection

Threshold detection, compared to the base load profile, is one of the methods to detect non-technical losses and electricity theft [40]. This example highlights a use-case for detecting atypically high electricity consumption by utilizing Hadoop MapReduce framework. The Hadoop job scans the data to count the number of times the power consumption was higher than a threshold. The data used in this example is regenerated and anonymized (for privacy, and confidentiality reasons) from real world smart meter data records accessed via the Irish Social Science Data Archive [41], where the readings were taken every 30 minutes from 5000 smart meters for a period of two years.

1) Hadoop Job: We consider a four-node Hadoop cluster in a standard data center network architecture as shown in Figure 1.

The objective is to count the total number of instances when the smart meter with ID 1400 reported the power consumption exceeding a baseline of 0.5 for the following day and time codes: 11518, 35010, 00120, 20513.[1]

A MapReduce task usually splits the input into independent chunks which are first processed by the *mappers* placed across different data center nodes in a completely parallel manner. The

[1]Each smart meter reading consists of a meter ID, a day and time code specifying the day of the year as well as the time—interval—of the day, and the power consumed.

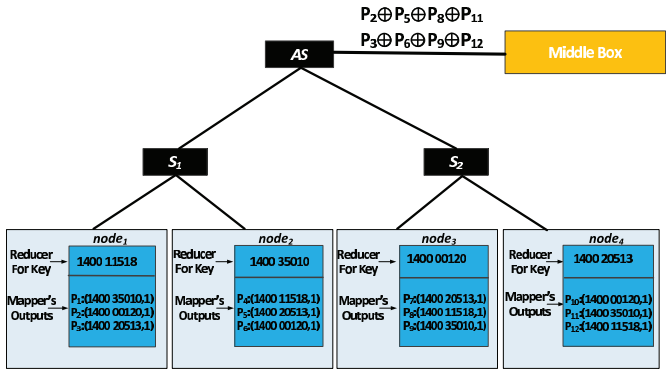

Fig. 1. Topology and mapper/reducer output/input in the electricity theft detection example.

outputs of the *mappers* are then communicated to the *reducers* which are also placed across different nodes, during the *shuffle* phase, for further processing to complete the job.

The input log of smart meter records is divided by Hadoop into four splits ($split_1, \cdots, split_4$). A split is stored on one of the four *DataNodes* ($node_1, \cdots, node_4$), whereby we assume that $split_i$ is stored on $node_i$.

Each $mapper_i$ residing on $node_i$ parses file $split_i$ and emits the corresponding $< key, value >$ pairs, where a *key* is the smart meter ID along with associated day and time code, and a *value* is the number of times the power consumption exceeds 0.5. In this job the role of the map function is to only extract the parts of the smart meter data log containing the information about the meter ID 1400 for one of the four targeted times and dates.

Each reducer ($reducer_i$ on $node_i$) processes the ($< key, value >$) pairs emitted by the *mappers* and counts the total frequency where power consumption exceeds the threshold. Each reducer is responsible for counting the ($< key, value >$) pairs corresponding to a single *key* as shown in Figure 1.

During the $shuffle$ phase the mappers outputs ($P_1, \cdots, P_{12}$ as shown in Figure 1) are delivered to the corresponding reducers, e.g., the mappers outputs with the key *1400 11518*—P_4, P_8, and P_{12}—are delivered to the $reducer_1$. Without loss of generality, we assume that a $< key, value >$ pair is communicated by a mapper to a reducer through a single packet transmission.

2) Standard Mechanism: Using standard Hadoop mechanism, it is easy to find that a total of 10 link-level packet transmissions are required per reducer to fetch all of its desired $< key, value >$ pairs from respective mappers. For example $reducer_1$ residing on $node_1$ is responsible for counting the $< key, value >$ pairs with all keys equal to "1400 11518", this process results in 10 link-level packet transmissions.

It follows then due to symmetry that 40 link-level packet transmissions are required to complete the *shuffle* phase. Note that a total of 16 link-level packet transmissions cross the network bisection ($AS - S_1, AS - S_2$) during the *shuffle* phase.

3) Proposed Coding-Based Mechanism: We note that in a typical real world Hadoop cluster a node hosts mutiple mappers and reducers [42], hence a reducer residing on the same node with a mapper has access to this mapper's output—referred as *side information*—without going through the network.

Leveraging on this *side information*, we propose coding the packets, whereby the coding refers to applying a simple $XOR(\oplus)$ function to a set of input packets. In this example, the coding is employed at the bisection, using middlebox attached to the $L2-aggregate$ switch AS as shown in Figure 1. The coding results in only two packets $P_2 \oplus P_5 \oplus P_8 \oplus P_{11}$, and $P_3 \oplus P_6 \oplus P_9 \oplus P_{12}$ crossing the bisection. Each reducer can use its *side information* to decode its required packets from the coded packet it receives as explained below.

Information-Theoretic Equivalence Relation: To explain the decoding process in an intuitive way, we introduce an *information-theoretic equivalence relation*, denoted by $\equiv$. Specifically, two different packets are *information-theoretically equivalent* if they convey the same information, for example although packets P_4,P_8, and P_{12} are different packets but as they carry the same information *(1400 11518,1)*, so we call them information-theoretically equivalent represented by an equivalent A, i.e., $P_4 \equiv P_8 \equiv P_{12} := A$. Similarly, $P_1 \equiv P_9 \equiv P_{11} := B$, $P_2 \equiv P_6 \equiv P_{10} := C$, and $P_3 \equiv P_5 \equiv P_7 := D$. Let's explore how $reducer_1$ would be able to decode its required packet fetched from $mapper_3$. We start by focussing on the coded packet $P_2 \oplus P_5 \oplus P_8 \oplus P_{11}$ received by $reducer_1$:

$$P_2 \oplus P_5 \oplus P_8 \oplus P_{11} \equiv C \oplus D \oplus A \oplus B$$

Then utilizing *side information* available at $reducer_1$:

$$\begin{aligned}(P_2 \oplus P_5 \oplus P_8 \oplus P_{11}) \oplus P_1 \oplus P_2 \oplus P_3 \\ \equiv C \oplus D \oplus A \oplus B \oplus B \oplus C \oplus D\end{aligned}$$

$$= A \equiv P_{11} = (1400\ 11518, 1),$$

i.e., the packet required by $reducer_1$ from $mapper_3$.

Together with the packet exchange occurring via the access switches and the transmission of the packets input to the coding function for them to reach the point of coding (aggregation switch in our example), we find that in this case a total of 36 link-level packet transmissions are required to complete the *shuffle* phase.

Note that by using coding a total of 12 link-level packet transmissions cross the network bisection during the *shuffle* phase, i.e., compared to baseline Hadoop implementation a 25% reduction in network bisection traffic. So our approach compared to current state of the art, depending on the use-case, **translates to 25% less energy utilization of the equipment, 25% more Hadoop jobs that run simultaneously, or to a 25% decrease in job completion time if there is congestion.** The use of identical values for each distinct key generated by a mapper in this example — picked deliberately to ease presentation — favours the efficiency of coding, but obviously may not hold for production Hadoop computations. We generalize the concept of *the coding-based shuffle* beyond this simplifying assumption in Section A.

B. Storm for DNA Sequencing

This example introduces the concept of *partial coding*, and shows how coding can be generalized to various $< key, value >$ pair patterns, and is oblivious to the semantics. In this example, we focus on Storm (an event processor) [15], a distributed real-time computation framework for processing unbounded data streams. Apache storm is an emerging big data platform used by Taoba, Ooyala, Infochimps, Weather channel, and Groupon.

$$\begin{aligned}P_1 &= (11, GCTTTAGCCGACCTGAACT.GACTACA)\\ P_2 &= (13, AGCTAGTCCCGAAGAAAATCTAGGTGG)\\ P_3 &= (12, ATAAGCATCAATACCACTAATATAGAT)\\ P_4 &= (10, GAGGTGATTGTGGTATTGT.GGTAAAT)\\ P_5 &= (42, ATAAGCATCAATACCACTAATATAGAT)\\ P_6 &= (23, AGCTAGTCCCGAAGAAAATCTAGGTGG)\\ P_7 &= (33, AGCTAGTCCCGAAGAAAATCTAGGTGG)\\ P_8 &= (20, GAGGTGATTGTGGTATTGT.GGTAAAT)\\ P_9 &= (21, GCTTTAGCCGACCTGAACT.GACTACA)\\ P_{10} &= (32, ATAAGCATCAATACCACTAATATAGAT)\\ P_{11} &= (41, GCTTTAGCCGACCTGAACT.GACTACA)\\ P_{12} &= (30, GAGGTGATTGTGGTATTGT.GGTAAAT)\end{aligned}$$

Fig. 2. Twelve records emitted by spout for the DNA-sequence processing.

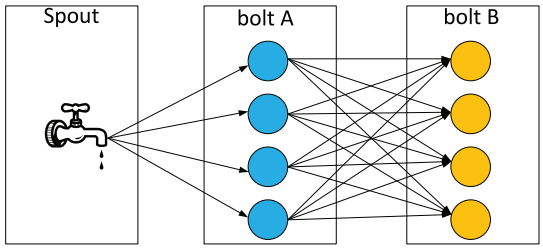

Fig. 3. Storm Topology for DNA sequencing example.

The core abstraction in Storm is the *stream* which is a continuous sequence of tuples. The basic components of Storm for the provision of stream transformations are *spouts* and *bolts*. A *spout* is a source of streams, whereas a *bolt* processes a number of input streams and emits the transformed streams. *Spout* and *bolt* tasks are spread across the Storm cluster.

To demonstrate the versatility of our proposed solution, we consider a network architecture that is different than the one considered in the Example A. More specifically, we consider a network architecture that is associated with Split multi-link trunking (IEEE 802.3ad) [43], NIC (network interface controller)-teaming with adaptive load balancing (ALB) [44], and load balancing on servers where different virtual machines assigned to different NICs.

1) Storm Job: The Storm job considered is a DNA-sequencing code, which processes samples of short DNA sequence records. Each DNA-sequence record consists of following two parts: a short DNA sequence (*value*), and a unique ID associated with this sequence (*key*). The objective of this job is to cluster short DNA-sequences based on the second digit of their unique ID. For demonstration purposes, we assume that the input data for this DNA-sequence processing job contains only twelve records as given in Figure 2.

Figure 3 shows the Storm topology for the DNA-sequencing job. The topology consists of one *spout*, and two *bolts*, *bolt A* and *bolt B*, where each bolt has four tasks.

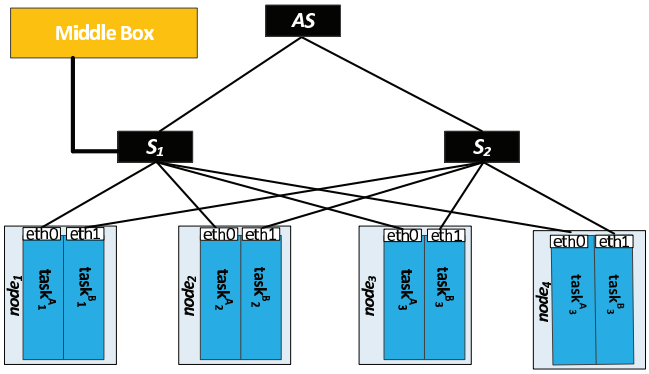

Fig. 4. Placement of tasks of *bolt A* and *bolt B* in the DNA sequencing example on a four node Storm cluster.

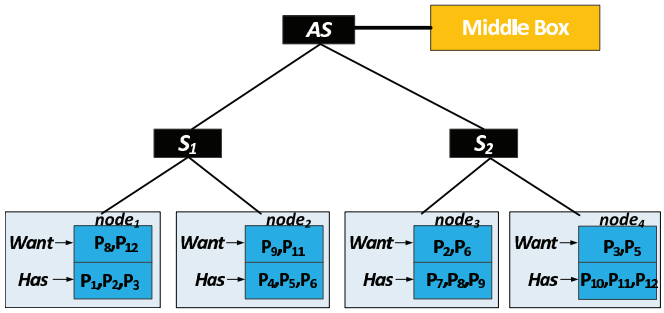

Fig. 5. An instance of *spate coding* for the example A.

The output from the spout is fed to the four tasks of *bolt A*. The *bolt A* emits modified streams through NIC $eth0$. These modified streams are fed to the *bolt B*, via NIC $eth1$, which is also running four tasks. The tasks of *bolt A* and *bolt B* are running on the four *DataNodes* $(node_1, \cdots, node_4)$ as shown in Figure 4; $task_i^k$ represents the task of *bolt k* running on $node_i$. Furthermore, the *stream grouping* for *bolt A* is *shuffle grouping* [45], which distributes the tuples to its tasks in a random fashion. The *stream grouping* for *bolt B* is *fields grouping*, which ensures clustering the short DNA-sequences based on the second digit of their unique ID. Specifically, the tuples from output streams of *bolt A* with the second digit of the key i will be delivered to the $task_{i+1}^B$. Without loss of generality, we assume that a $< key, value >$ pair is transmitted from *bolt A* to *bolt B* through a single packet transmission, and the spout's output is such that $P_1, \cdots, P_3$ are received by $task_1^A$, $P_4, \cdots, P_6$ are received by $task_2^A$, $P_7, \cdots, P_9$ are received by $task_3^A$, and $P_9, \cdots, P_{12}$ are received by $task_4^A$.

2) Standard Mechanism: We proceed to analyze this scenario from the perspective of the standard Storm mechanism. Assuming each link to have the same capacity, it is easier to see that the links on the path from switch S_1 to S_2 i.e., $S_1 - AS - S_2$, are the bottleneck links. Calculating the total link utilization incurred by a $task_i^A$ to communicate the transformed streams to the corresponding tasks of *bolt B*, we easily find that a total of 12 link-level packet transmissions are required using standard Storm mechanisms. It follows then due to symmetry that 48 link-level packet transmissions are required to complete the transfer from all tasks of *bolt A* to the corresponding tasks of *bolt B*. Note that a total of 24 link-level packet transmissions cross the network bisection.

3) Proposed Coding-Based Mechanism: By observing the output tuples from the tasks of *bolt A*, it is obvious that in contrast to Example 1, coding on the basis of an entire $< key, value >$ pair will be of no advantage in this example. This follows from the fact that each of the $P_1, \cdots, P_{12}$ has a unique Key, hence no task of the *bolt B* has sufficient *side information* to decode a subset of entire packets coded together. Therefore, we exploit the novel concept of *partial (a.k.a. fractional) coding* . At S_1, the L2-Switch of the toy network topology in Figure 4, the coding is performed only on the portions of the packets (second digit of the key and complete value). More specifically, the coded packets are: $\Omega_1 \oplus \Omega_4 \oplus \Omega_7 \oplus \Omega_{10}$, $\Omega_2 \oplus \Omega_5 \oplus \Omega_8 \oplus \Omega_{11}$, and $\Omega_3 \oplus \Omega_6 \oplus \Omega_9 \oplus \Omega_{12}$, where Ω_i represents part of P_i comprising the second digit of the key and complete value; for example Ω_1 is highlighted in red block in Figure 2. Each of these coded data are then combined with their corresponding raw (i.e. not coded) portions to form a partially coded packet. Further details on the format of a partially coded packet are given in Section A.

Following similar arguments used in Section A, it can be shown that each task of the *bolt B* can decode its required $< key, value >$ pairs. It is interesting to note that the *coding-based* network transmissions results in significantly—namely 75%—less utilization of network bisection links, while maintaining the same amount of information transfer as without coding. Depending on the use-case, this translates to 75% more Storm jobs running simultaneously, or to a 75% decrease in job completion time if the links between $AS - S_1$ and $AS - S_2$ are congested.

III. Spate Coding Problem

This section formalizes the concept of coding by defining *spate coding* problem. While *spate coding* has a similar flavour to network coding and index coding, Appendix IX highlights the differences, and show that *spate coding* problem subsumes both the network coding and index coding problems.

Definition 1 (Spate coding problem): An instance of the *spate coding* problem is defined by a coding server, a set $X = \{c_1, \ldots, c_m\}$ of m clients (nodes), and a set $P = \{p_1, \ldots, p_n\}$ of n packets that need to be delivered to the clients. Each client c_i is interested in a certain subset of packets known as its *Want* set $W_i \subseteq P$, and has access to some *side information* known as its *Has* set H_i. Note, a client might not possess any *side information*, i.e., its *Has* set might be empty. The server can transmit the packets in P, or their combinations (coded packets) to the clients via a point to point network that supports multicast. For efficiency, the only coding operations that the server is required to perform are restricted to $\oplus$ i.e., operations over $GF(2)$. The goal is to find a transmission scheme that requires the minimum number μ link-level packet transmissions to satisfy the requests of all clients.

The example in Section A can be represented by an instance of *spate coding* problem where a coding server, co-located with $L2 - Aggregate$ switch (AS), needs to deliver eight packets $P = \{P_2, P_3, P_5, P_6, P_8, P_9, P_{11}, P_{12}\}$ to a set of four clients

$X = \{reducer_1, reducer_2, reducer_3, reducer_4\}$. Demand of the $reducer_1$ is given by its *Want* set $W_1 = \{P_8, P_{12}\}$. The *side information* available to the $reducer_1$ is given by its *Has* set $H_1 = \{P_1, P_2, P_3\}$, and similarly for rest of the reducers as shown in Figure 5). We present the proposed solution to *spate coding* problem in Section VII.

IV. Challenges and Considerations

Some of the obvious questions with regards to our proposal for the use of mixing (coding) are:

1. How does the *coder* (middlebox) collects the *side information*, and determines destination of each coded packet?
2. How is *spate coding* problem incorporated in the proposed architecture?
3. What is coding format, and how to determine which parts of $< key, value >$ pairs should be encoded?
4. How to encode the packets while being able to perform practical line-rate processing?
5. What extra information is required in each packet, in addition to a packet's payload, so that each client can decode the required packets instantaneously.
6. How does a client decodes a packet?
7. Can we say something about computational complexity of such schemes in general, and provide some bounds on the advantage?
8. Can the proposed architecture integrate seamlessly in the current big data architectures (like Hadoop)?
9. How does the proposed scheme perform in practice?

We present a complete architecture that encompasses questions 1 to 6 in Section A. We present some analysis related to question 7 in Section V. The answer to question 8 is discussed in Section C, and regarding question 9 we present evaluation results from a clustered prototype implementation of proposed scheme in Section VIII.

V. Analysis

In this section we analyze the computational complexity and advantage of the schemes that tend to minimize data transmission in distributed data center applications.

Let a scheme S_i results in a total number of $|P(S_i)|$ link-level packet transmissions during the communication phase of distributed data center application running over a network $\mathbb{N}$. For example in Hadoop this communication phase is called *shuffle* phase. We start by defining the problem $\mathbb{ES}$.

Definition 2 (Problem $\mathbb{ES}$): Find a scheme S such that for all other schemes S_i: $|P(S)| \leq |P(S_i)|$.

Let $OPT(S)$ denote the optimal solution to the problem $\mathbb{ES}$.

Theorem 1: The Problem $\mathbb{ES}$ is NP-hard, and is NP-hard to approximate as well.

The proof of Theorem 1 is given in A.

We begin to analyze the maximum and minimum advantage of the proposed coding based scheme can offer compared to the current standard non-coding (routing) based techniques. We start by defining the *Utilization Ratio*.

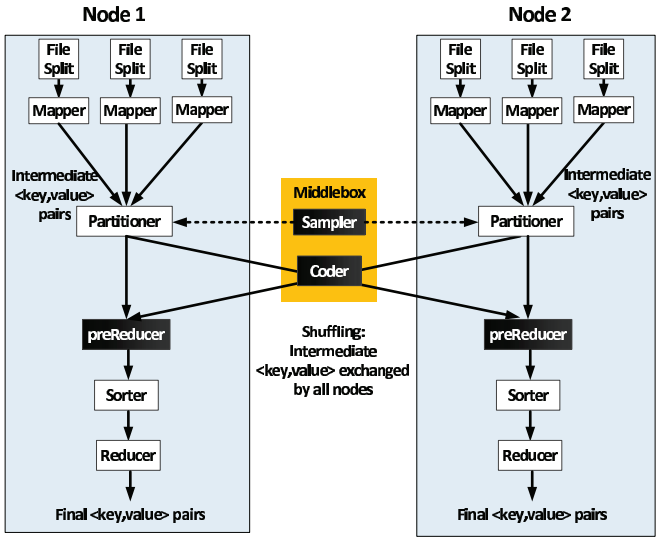

Fig. 6. The proposed coding based Hadoop MapReduce data flow.

Definition 3 (Utilization Ratio (μ)): Utilization Ratio μ is the ratio of link-level packet transmissions when employing a coding based solution to the number of link-level packet transmissions while not employing a coding based solution.

We also define diameter of a network as follows.

Definition 4 (Diameter of a Network (d)): The diameter of a network d is the longest of all the shortest paths (in terms of number of hops) between DataNodes in a network.

Theorem 2 provides a lower bound on the utilization ratio for a general network in terms of its diameter d.

Theorem 2: $\mu \geq \frac{1}{d}$.

The proof of Theorem 2 is given in D.

Corollary 1: For distributed data center applications running on large number of nodes (larger than the network diameter), $\mu \geq \frac{1}{\text{number of nodes}}$.

Theorem 3: $\mu \leq 1$, and this bound is a tight bound.

The proof of Theorem 3 is given in E.

Next, we analyze the maximum advantage a coding based scheme can offer with reference to a network's bisection when the coding is also performed at the network's bisection. We proceed by defining Utilization Ratio with reference to a network's bisection.

Definition 5 (Bisection Utilization Ratio ($\mu(bisection)$)): Bisection Utilization Ratio $\mu(bisection)$ is the ratio of link-level packet transmissions crossing the network's bisection when employing a coding based solution to the number of link-level packet transmissions crossing a network's bisection while not employing a coding based solution.

Theorem 4 provides an upper bound on $\mu(bisection)$ while is also performed at the network's bisection.

Theorem 4: $\mu(bisection) \geq \frac{1}{2}$, while coding is also performed at the network's bisection, and this bound is a tight bound.

The proof of Theorem 4 is given in F.

Corollary 2: Network Bisection is not an optimal location to place the middlebox for performing coding.

The proof of Corollary 2 is given in G.

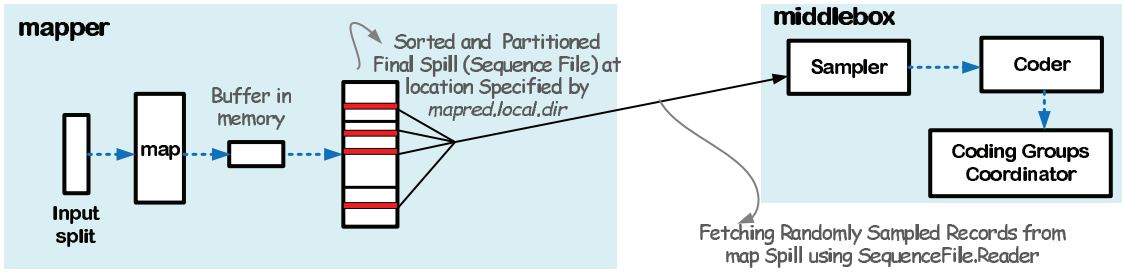

Fig. 7. Architecture for Middlexbox (*sampler*, and *coder*).

VI. Theory to Practice

A. Coding-Based Middlebox and Its Components

For ease of explanation as well as to be in accordance with our proof of concept prototype implementation, we present the details of our architecture with reference to Hadoop Mapreduce. It can be easily extended to other distributed data center applications. Moreover, the proposed coding scheme is independent of the underlying application. We introduce three new stages, namely *sampler*, *coder*, and *preReducer* to the traditional Hadoop MapReduce. The primary objective of the sampler is to gather *side information*. Similarly the primary objective of the coder is to code, and of the preReducer is to decode. The overall architecture is shown in Figure 6, while it shows only two nodes it is in fact replicated across all the nodes.

In this section we focus on the middlebox, and its components (*sampler*, and *coder*).

1) Sampler: The *sampler* gathers the *side information*. Specifically, the *sampler* resides in the middlebox and fetches $\aleph$ random records from the mappers at each of the physical machines. These random records are fetched in parallel to the *shuffle* phase. This sampling process does not interfere with the *shuffle* process. These sampled records not only help to start building the *side information* available at each node but also play a pivotal role in the decision of which portions (segments) of the packets should be coded (please refer to the Example in Section B regarding partial coding).

The overall data flow from a mapper to the *sampler* is shown in Figure 7. Specifically, a mapper first completes emitting the $< key, value >$ pairs to the intermediate spills. These intermediate spills are then merged to form a final *sorted and partitioned* sequence file. In the final sequence file each partition corresponds to a particular reducer. The *sampler*, co-residing with the *coder*, fetches the random records from the final sequence file which is stored in *SequenceFile* format at the locations specified by *mapred.local.dir* [42]. The records from the sequence file is read using java class *org.apache.hadoop.io.SequenceFile.Reader* [46]. These sampled $< key, value >$ pairs serve as the initial *Has* set for each physical machine.

2) Coder: The *coder* is a dedicated software appliance, strategically placed in the middlebox, for wire-speed packet payload processing (typically XORing packet payloads) and re-injection of coded packets into the network. Specifically, the *coder* initially receives $\aleph$ inputs from the partitions of all mappers via the *sampler*. After that it only receives the information that passes through the part of the network where it is placed. The *coder* performs the following three functions:

(1) ***Format Decision Making:*** Based on the sampled data records the *coder* decides on a coding format consisting of byte indices ω_1, ω_2. The *coder* treats a $< key, value >$ pair as a *data chunk*. In a *data chunk* the bytes starting from index ω_1 and ending at index ω_2 are the ones that are anticipated for coding for the following $\aleph_c$ generations; we name these bytes the *encodable chunk*. A *generation* specifies a group of packets to be processed together by the *coder*. The rest of the bytes in the *data chunk*, named the *uncodable chunk*, are forwarded without coding. The logic behind choosing ω_1, ω_2 is to find partitions of the *data chunk* that maximize coding advantage, as for a specific Hadoop job some bytes of $< key, value >$ pairs might contain more mutual information than others. Note that coding exploits the mutual information between different file splits residing on different physical machines. For the example in Section B, if each digit in the $< key, value >$ pair represented by a byte then $\omega_1 = 2, \omega_2 = 28$, and the *uncodable chunk* is just the first byte and the rest of the packet is the *encodable chunk*.

The decision on the coding format is based on comparing the coding advantage over the $\aleph$ random records fetched by the *sampler* from different mappers. Coding advantage is evaluated for different values of ω_1, ω_2. The computational complexity of determining the best contiguous *encodable chunk* is logarithmic (a very efficient procedure based on binary partition). The complexity of finding arbitrary number of noncontiguous *encodable chunks* can result in higher coding advantage, but can also increase the computational complexity, more book-keeping, and a more complex packet header.

The coding format is periodically re-computed after every $\aleph_c$ generations to fine tune the decision based on particularities of the Hadoop job.

(2) ***Coding:*** This step performs binary coding (bitwise XOR) based on the bytes ω_1 to ω_2 (*encodable chunk*) from each generation of received $< key, value >$ pairs. The coding algorithm is based on solving the *spate coding* problem in an efficient fashion, and is presented in Section VII. We note that the algorithm requires information about both the *Want* set and *side information (Has set)* of each node in the cluster participating in the Hadoop job under consideration. In our scheme, the *Want* set is determined based on the key, whereas knowledge about the *Has* set keeps on building at each generation. During each generation the new *side information* is extracted

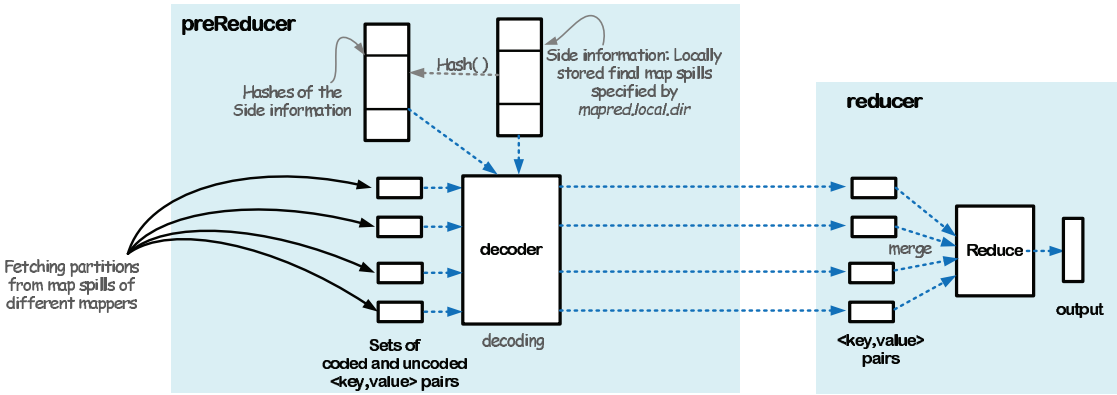

Fig. 8. Architecture for *preReducer*.

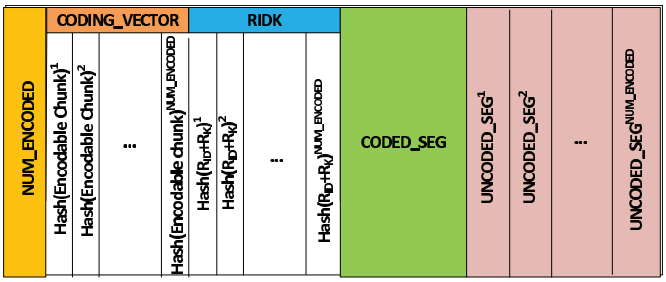

Fig. 9. The proposed packet payload format for the coding based shuffle at the application-layer. Depending on size, the format shown may be transported using multiple segments/packets.

from the bytes ω_1 to ω_2 from the $< key, value >$ pairs available at the *coder*. For the coding purpose, we maintain the *Has* set of all the nodes without any assumption over the extent of information overlaps between different physical machines. The information overlaps between different physical machines can be arbitrary or none. Moreover, the coding can result in multiple multicast packets to be forwarded to set of *reducers*. Each multicast packet is computed for the subtree of the network topology under the *coder* (parent) node. To enable processing at the line-rate, the algorithm ensures that the coded packets are *instantaneously decodable* at the receiver nodes i.e., just based on the packets received and without any need of buffering a set of incoming packets.

(3) ***Packaging:*** This step packs the outcome of the coding process into a custom packet payload format, as shown in Figure 9, that is then re-injected into the network towards the reducer nodes. Different fields in the packet payload format collectively ensure that each reducer finally receives the $< key, value >$ pairs intended for it. The fields are:

- $NUM\_ENCODED$: It is a numerical value describing the number of packets that are coded together. For instance, for the packet $p_2 \oplus p_5 \oplus p_8 \oplus p_{11}$ of Example in Section A, this value is 4. In case where no encoding is performed, this field is set to 0.
- $CODING\_VECTOR$: It is a vector of size $NUM\_ENCODED$, and contains *hashes* of the *encodable chunks*. There can be a large set of *encodable chunks*, so we use *hashes* for their fast matching with the *Has* set to identify *side information* for the decoding process given in Section B.
- $RIDK$: It is a vector of size $NUM\_ENCODED$, and specifies intended reducers along with its corresponding key. We associate an ID (a bit vector) R_{ID} with the reducer R, and R_K denotes the key assigned to the reducer R. $RIDK$ contains *hashes* of R_{ID} R_K pairs. Moreover, a reducer might work on multiple keys, and does not know in advance what $< key, value >$ pair to expect in the received packet, so it is necessary that a packet should contain information about the intended reducers and its corresponding key.
- $CODED\_SEG$: It is the actual *coded* chunk, formed by bitwise XOR of the $NUM\_ENCODED$ *encodable chunks*.
- $UNCODED\_SEG$: It contains $NUM\_ENCODED$ *uncodable chunks*.

B. PreReducer

The major role of the *preReducer* component is to ingest the custom-made packets sent by the *coder*, decode their payloads and extract the $< key, value >$ pairs which are to be input to the standard Hadoop *Sorter*.

We recall that a mapper stores the output $< key, value >$ pairs in a set of partitions, where each partition contains $< key, value >$ pairs for a particular reducer. We further note that based on the placement of the middlebox, or some routing intricacies some of the $< key, value >$ pairs fetched by a reducer might not pass through the middlebox. Hence, *preReducer* should tackle both types of the packets, namely packets coming through the middlebox with additional header, and packets not-coming through he middlebox. Specifically, the *preReducer* passes the packets not-coming through the middlebox to the reducer without any changes. Whereas, the *preReducer* performs decoding, and $< key, value >$ extraction process on the packets coming through the middlebox. The complete architecture for the *preReducer* is shown in Figure 8.

The *preReducer* decodes the coded part of the packet based on the coding format, and forms $< key, value >$ pairs by inserting the decoded chunk from byte index ω_1 to ω_2 into

the corresponding *uncoded chunk* of the packet. Once a node receives a packet, the *preReducer* first checks the $RIDK$ field to determine the intended reducer and the corresponding key. Then it identifies the *side information*, required to decode its intended *encodable chunk*, by comparing the hashes of the mappers outputs stored locally with the hashes contained in $CODING\_VECTOR$. The decoding is performed by $XORing\ CODED\_SEG$ with the *side information*. The detailed decoding process is given in in Algorithm 1.

Algorithm 1. PreReducer

```
n = NUM_ENCODED
if n ≠ 0 then
    Retrieve the index myIndex from the packet by sifting through the RIDK vector
    decoded_Seg = CODED_SEG
    for i = 1 to n do
        if i ≠ myIndex then
            has = lookup(Hash^i)
            decoded_Seg = decoded_Seg ⊕ has
        end if
    end for
    Retrieve < key, value > by inserting decoded_Seg at index ω1 + 1 in the UNCODED_SEG at index myIndex
else
    < key, value > is same as packet received
end if
```

C. Seamless Integration Using OpenFlow

For a seamless integration of the proposed scheme into the Hadoop architecture, we propose a novel software-defined-networking, specifically OpenFlow, based scheme for multicasting coded packets. Unlike conventional receiver-initiated multicast, the use of OpenFlow (version 1.2) provides the ability to have the middlebox control the dynamics of multicast groups on-demand. Hadoop contains information about the placement of mappers and reducers in the cluster, and the topology they are connected through [47]. Using this information mappers and reducers can be grouped into subtrees.

The proposed scheme implements multicast state across a multicast group G—at each switch along the tree topology—using a *Group Table* entry with *Group ID=G*. The *Group Type* is set to *all (multicast)*, and respective *Action Bucket* is set to forward the packets to the set of destination ports corresponding to the multicast tree links. We overload here the semantic of two IP header fields, namely the *ECN* and *DS* fields (8-bits long in total), to match multicast group state on a switch with packets of a specific multicast group, allowing us to use 256 multicast groups in total. An alternative is to use *Locally Administered* multicast MAC addresses, which can allow handling of upto 2^{46} multicast groups.

The following two practical constraints are associated with the implementation of multicast in large-scale data centers. First, keeping a multicast group for every coding combination, determined by the *coder*, may be prohibitively expensive. Second, creating mulitcast groups on-demand can incur high-latency. To meet these two practical constraints, we intorduce a component called the *coding groups coordinator (CGC)* within the *coder*. The *CGC* coordinates, in tandem with the OpenFlow controller, the dynamic creation of multicast groups using a small (constant) working set of multicast groups. Specifically, the *CGC* maintains at most α (a small constant) group-queues. Each group-queues is associated with a distinct set of reducers (group of receivers) and being identified by a unique *Queue-ID*. In addition to this, the *CGC* maintains one *general queue* that is drained to the network. Packets from the *coder* destined to the same multicast group are buffered in the same group-queue, whereby at the event of adding the first packet to a group-queue, the *CGC* notifies the OpenFlow controller to create network state for the multicast group that the packet corresponds to. Once a queue is fully flushed, the *CGC* picks the next multicast group from the *general queue* and assigns the emptied queue to a new multicast group, while in parallel the next group-queue is drained to the network. This ensures a temporal overlap of multicast state creation for a future multicast group while also operating on a fixed budget of multicast state (#groups). In short, our approach somewhat resembles with the concept of virtual queues in network routers, without though the constraint of starvation, for in this case the workload does not pose deadlines on identical virtual queues due to Hadoop being throughput and not latency-bound.

Figure 10(b) describes the process of multicasting four encoded packets $P_2 \oplus P_5$, $P_3 \oplus P_6$, $P_8 \oplus P_{11}$, and $P_9 \oplus P_{12}$, found as a result of execution of Algorithm 2 applied on the example in Section A. Figure 10(a) shows the entries for *Flow Tables* and the corresponding *Group Tables* for each switch in the multicast tree. In particular, the *CGC* creates two queue, the queue with *Queue-ID=1* for the multicast group consisting of $node_1$ and $node_2$ (destination of $P_8 \oplus P_{11}$ and $P_9 \oplus P_{12}$), and the queue with *Queue-ID=2* for the multicast group consisting of $node_3$ and $node_4$ (destination of $P_2 \oplus P_5$ and $P_3 \oplus P_6$). The *CGC* then creates the network state for the these multicast groups. For all the packets in the queue with *Queue-ID=1* the *DS+ECN* field is set equal to 1, and for all the packets in the queue with *Queue-ID=2* the *DS+ECN* field is set equal to 2. The *Flow Table* entries and corresponding match-actions inspects the *DS+ECN* field of a packet, and then utilize *Group Table* to find the corresponding multicast group and then forward based on the *Action Bucket*. For example, the packets with *DS+ECN* field equal to 1 are forwarded according to the *Action Bucket* under *Group ID* 110, i.e., multicasted to the ports a and b of the switch S_1, and port x of the switch AS.

VII. Solution for Spate Coding Problem

We start by defining the *dependency graph* to capture mutual information between the clients (nodes).

Definition 6 (Dependency Graph): Without loss of generality, for the ease of presentation, we assume each client requires just one packet. In case a client χ requires more than one packet say $|W|$ packets, we replace it by $|W|$ clients each having one of the $|W|$ packets as its *Want* set, and all the $|W|$ clients have the same *Has* set as of χ. Given an instance of *spate coding*

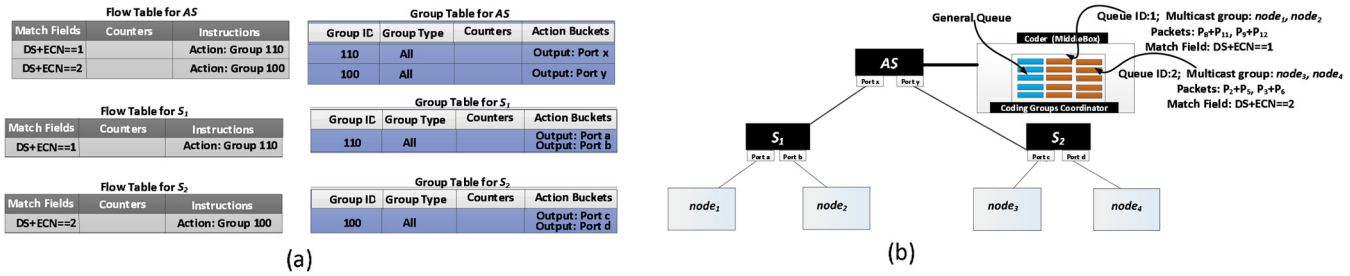

Fig. 10. (a) *Flow Tables* and the corresponding *Group Tables* for each switch in the multicast tree for the example given in section A.(b) OpenFlow based multicasting using *coding groups coordinator* for the example given in section A.

Algorithm 2. ComputeCode

Input: An instance of *spate coding* problem

1: $X' := \phi$
2: **while** $X \neq \phi$ AND $X \neq X'$ **do**
3: For each set of clients $\hat{X} = \{c_1, c_2, \cdots, c_y\}$ such that $W_1 \equiv W_2 \cdots \equiv W_y$, $X' := X \setminus \{\hat{X} \setminus c_1\}$
4: From the given instance of *spate coding* problem for the clients in X', construct the *dependency graph* $G(V, E)$;
5: **for** $k = \lambda$ down to 2 **do**
6: **while** $\exists$ a clique ϑ in $G(V, E)$ of size k **do**
7: Divide clique into x smaller cliques θ_i each belonging to different subtrees in the multicast topology.
8: **for** $i = 1$ to x **do**
9: Multicast a packet that satisfies all clients corresponding to the vertices in θ_i;
10: $V := V \setminus \theta_i$;
11: Delete the clients corresponding to the vertices in the θ_i from the client set X
12: **end for**
13: **end while**
14: **end for**
15: **end while**
16: **if** $X \neq \phi$ **then**
17: Send the uncoded packets corresponding to the clients in X
18: **end if**

problem, we define a *dependency graph* $G(V, E)$ with vertex set V and edge set E as follows:

- For each client c_i there is a vertex v_i in $G(V, E)$;
- For any two clients c_i and c_j not residing on the same node (server), there is a directed edge, in $G(V, E)$, from v_i to v_j if and only if it holds that for $P_i \in W_i, \exists\, P \in H_j$: $P \equiv P_i$, where $\equiv$ represents the information-theoretic equivalence relation described in Section A

Figure 11 shows the *dependency graph* for the Example in Section A.

Lemma 1: A clique of size n in the *dependency graph* represents a a group of n clients, whose requests can be satisfied with only one coded packet.

Proof: Follows from the fact that all the clients represented by a clique in the *dependency graph*, can be satisfied by one transmission which consists of an $\oplus$ of all the packets in their *Want* sets. ■

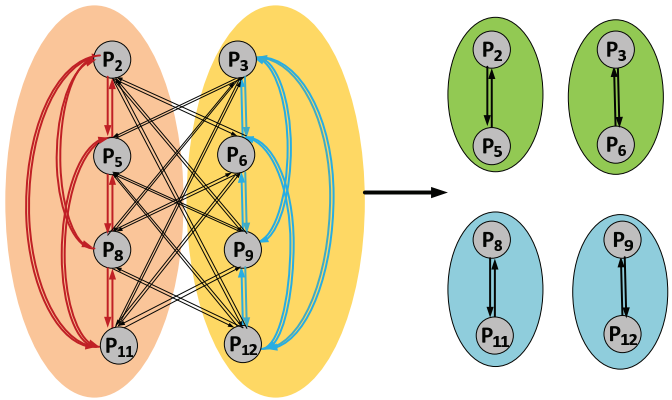

Fig. 11. *Dependency graph* for the transmissions crossing *AS* for the Example given in A. The execution of the Algorithm 2 results in four multicast packets shown by four cliques each of size 2.

The proposed Algorithm 2, named *ComputeCode*, greedily packs as many cliques as possible starting from the largest clique of size λ. Each clique corresponds to only one multicast packet, therefore packing cliques starting from larger cliques heuristically ensures more savings in terms of number of transmissions. However finding all cliques can become computationally prohibitive, we therefore focus on finding all the cliques of size $\lambda = 4$ and less to allow close-to-runtime execution of the coding algorithm. Increasing λ could possibly increase coding benefit. However, in addition to time-versus-coding benefit tradeoff, another important factor to consider while selecting λ is the maximum number of leaf nodes in the subtree rooted at the middlebox. λ should not exceed the maximum number of leaf nodes in the subtree, since the maximum coding advantage is achieved for the case when all the packets heading to the leaf nodes are coded together as one packet.

The computational complexity of our clique packing algorithm is $O(|V|^4)$. In general, extending the same algorithm for finding all cliques of size λ and smaller results in $O(|V|^\lambda)$ computational complexity. We note that the computational complexity of clique finding can be significantly improved to $O(\kappa^{\lambda-2}|E|)$ using algorithms similar to [48], where κ is arboricity of the dependency graph. For graphs with small κ, e.g., *planar graphs* with $\kappa = 3$, clique finding is a very efficient operation of $O(|E|)$.

To reduce the overhead associated with each multicast packet, we exploit the relationship between a mutlicast packet,

corresponding multicast tree, and corresponding clique. We note each sub-clique, a subset of the vertices and corresponding edges of a clique, encodes fewer packets than the original clique. Moreover, a sub-clique corresponds to a sub-tree of the multicast tree associated with the original clique. Hence, we divide a clique in sub-cliques in a way such that each multicast sub-tree carries a packet that encodes a subset of the packets in the original encoded packet, resulting in less coding overhead. For instance for the packet format proposed for Hadoop in Section 2, the header of each multicast packet conveys the information regarding each of the packet in *CODED_SEG* using *CODING_VECTOR, RIDK, UNCODED_SEG*. So if *CODED_SEG* contains fewer packets the size of *CODING_VECTOR, RIDK, UNCODED_SEG* is smaller. Also note that since each sub-clique is also a clique so the validity of the solution holds i.e., each client can decode instantaneously without any need to buffer the packets it received. *Further, dividing a clique into smaller cliques does not increase the number of overall link-level packet transmissions as it is equivalent to separating information intended for different clients.*

The execution of proposed algorithm resulted in two cliques each of size 4 (shown in pink and orange in Figure 11) for the example in Section A. These cliques are further subdivided into two sub-cliques, one sub-clique is associated with the multicast sub-tree rooted at the switch S_1 (shown in green), and other one is associated with the multicast sub-tree rooted at the switch S_2 (shown in blue) in Figure 11. This corresponds to multicasting packets $p_2 \oplus p_5$ and $p_3 \oplus p_6$ to $reducer_3$ and $reducer_4$, and multicasting packets $p_8 \oplus p_{11}$ and $p_9 \oplus p_{12}$ to $reducer_1$ and $reducer_2$.

VIII. Performance Evaluation

We developed a prototype as well as a testbed to evaluate the performance of the proposed coding based approach. Section A describes the prototype and Section B describes the testbed. We want to emphasize that the prototype and the testbed depict two of the most commonly used real-world system development models i.e., proprietary–vs–open-source. The prototype was implemented in a data center using the costly proprietary tools, and hardware. Whereas the testbed was implemented using open-source tools, virtualized environments, and commodity off-the-shelf components. The experimental results showed the advantage of our proposed scheme in both the models. We use data from Hadoop shuffle to benchmark our proposed solution. The Hadoop jobs consisted of the following two industry standard benchmarks.

1) *Terasort*, a benchmark to sort 10 billion records (1 Terabytes (TB)). These records are generated using a program called *TeraGen* [49]. Each record consists of 100-bytes, with the first 10 bytes being the key and the remaining 90 bytes being the value. This benchmark represents the works loads from mathematical applications to application using artificial intelligence.
2) *Grep* (Global Regular Expression) represents generic pattern matching and searching on a data set. Our data set consisted of on an organization's data-logs, whereby the goal was to calculate the frequency of occurrence of eight different types of events in the data-log input. Applications of *Grep* vary from data mining and sequencing to anamoly detection.

We want to point out that these two benchmarks not only cover a wide range of big data applications, but also cover a wide spectrum of the network traffic generated by big data applications. In fact, Cisco has used the same two types of benchmarks to study big data infrastructure considerations, and performance of their proposed solutions. Furthermore, these two benchmarks capture the workloads with two very different traffic patterns, namely "Business Intelligence (BI)" workload benchmark captured by Grep, and "Extract, Transform, and Load (ETL)" workload captured by Terasort [50]. Moreover, these benchmarks are two of the most widely used standard practice benchmarks for assessing performance of the Hadoop MapReduce implementations [21], [29], [31], [51]–[53].

A. Prototype

We have prototyped *sampler*, *coder*, and *preReducer* (decoder) in a data center as an initial proof of concept implementation. Our testbed consisted of 96 cores arranged in 8 identical blade-servers. Each server was equipped with twelve x86_64 Intel Xeon cores, 128 GB of memory, and a single 1 TB hard disk drive. The servers were arranged in three racks. Furthermore, the servers were connected in a typical data center configuration (resembling the one shown in Figure 1) with OpenFlow enabled IBM RackSwitch G8264 as Top-of-Rack switches, and OpenFlow enabled Pronto 3780 as Aggregation switches. One server was used as the middlebox. The components were implemented using Java, and Octave [54]. All the servers were running Red Hat Enterprise Linux 6.2 operating system [55].

We use the following metrics for quantitative evaluation:

- *Job Gain*, defined as the increase (in %) in the number of parallel Hadoop jobs that can be run simultaneously with *coding based shuffle* compared to the number of parallel Hadoop jobs achieved by standard Hadoop
- *Utilization Ratio*, defined as the ratio of link-level packet transmissions when employing *coding based shuffle* to the number of link-level packet transmission incurred by the standard Hadoop implementation.

Our experimental study shows that for both of the tested benchmarks, the overhead to implement *coding based shuffle* (in terms of transmission of extra bookkeeping data in packet headers) was less than 4% in all the experiments. Table I shows the results across the two metrics for the two benchmarks. The results show significant performance of our scheme compared to the standard Hadoop implementation.

Noting the fact that our coding based scheme just requires $XORing$ of packets which is computationally very fast operation and given high memory bandwidth of the servers, we were able to process closer to line rate. Specifically, even during the worst case scenario, the throughput of the coder was 809 $Mbps$ on a 1 $Gbps$ link.

TABLE I
JOB GAIN AND *UTILIZATION RATIO* USING PROPOSED CODING-BASED SHUFFLE

Hadoop Job	Job Gain	Utilization Ratio
Sorting	29 %	.71
Grep	31%	0.69

B. Testbed

Our testbed consisted of eight virtual machines (VMs), each running CentOS 7 as the operating system [56], on top of x86_64 Intel i7 cores. CentOS is a free Linux distribution aimed at providing enterprise class functionality compatible with Red Hat Enterprise Linux [55]. We used Citrix XenServer 6.5 as the underlying hypervisor [57]. In addition to more than 1 TB of local hard disk and solid state drives, each VM had access to 3 TB of network attached storage. Citrix XenCenter was used to manage the XenServer environment and deploy, manage and monitor VMs and remote storage [58]. Open vSwitch [59] was used as the underlying switch providing network connectivity to the VMs. Open vSwitch is a production quality, distributed, multilayer virtual switch designed to enable massive network automation supporting OpenFlow as well as NIC-teaming. Rest of the software implementation was same as used in Section A.

Moreover, we have implemented a stand-alone split-shuffle, to provide better insights into shuffle dynamics, where receiver service instances (e.g., reducers) fetch file spills from sender service instances (e.g., mappers) in a split fashion using a http mechanism.

To investigate performance of the proposed scheme as well as middlebox placement in different scenarios, we used following two commonly-used data center topologies:

1) Tree topology (resembling the one shown in Figure 1) with middlebox at bisection. We denote this topology by Top-1.
2) Tree topology with NIC-Teaming (resembling the one shown in Figure 4). Moreover, in this topology the middlebox is placed at first L2-switch. We denote this topology by Top-2.

A sorting benchmark was constructed using concatenated Terasort records. We measured the following parameters of interest:

- *Volume-of-Communication (VoC)*, defined as the amount of data crossing the network bisection. Note that VoC and utilization ratio, although related to each other, measure two different quantities.
- *Goodput (GP)*, defined as the number of useful information bits delivered to the receiver service instance per unit of time.
- *%disk-utilization (%d-util)*, defined as the percentage of CPU time during which I/O requests were issued to the storage disk. Disk saturation occurs when this value is close to 100%. Higher %d-util means poor performance (disk bottleneck).
- *Queue-size (Qsz)*, defined as average queue length of the requests that were issued to the storage disk. Higher Qsz means longer wait time.

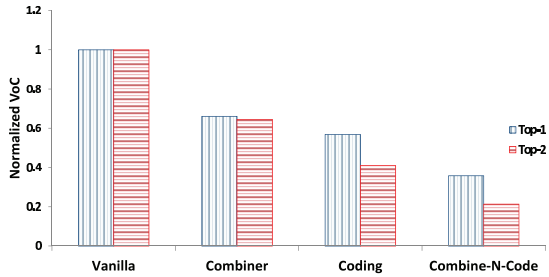

Fig. 12. Normalized *VoC* using *Grep* benchmark for both topologies Top-1 and Top-2.

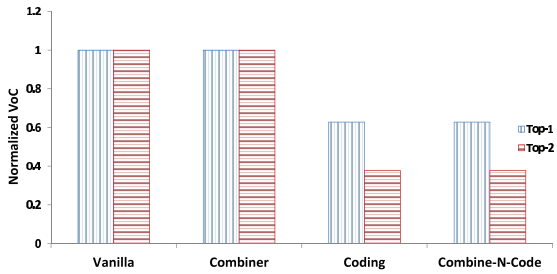

Fig. 13. Normalized *VoC* using *Sorting* benchmark for both topologies Top-1 and Top-2.

- *Bits-per-Joule (BpJ)*, defined as the number of bits that can be transmitted per Joule of energy.

The selected performance parameters are closely connected and more comprehensively express the benefits that the proposed scheme has to offer in a holistic fashion capturing both the network, and the nodes. Specifically, volume of communication, and number of bits transmitted per Joule of energy focus on the network aspects, whereas goodput, disk utilization, and queue size capture a node's perspective.

VoC measures overall data transfer (network load) during the shuffle phase. On the other hand, goodput measures the average *effective* throughput, crucial from an application's perspective. Furthermore, aside from the amount of data and data rate, the energy required to shuffle data is an important parameter indicating the level of greener optimization offered by the proposed scheme. Similar is the importance of measuring device utilization and queue size capturing the advantage of the proposed scheme in terms of availability of the system resources to different processes and virtual machines.

1) Volume-of-Communication: In this Section we compare *VoC* of our proposed approach with vanilla Hadoop and a state of the art in-network combiner. An in-network combiner reduces the *VoC* by partially distributing functionality of the receiver service instances over the network [29], [60]. Additionally, to demonstrate the complementary nature of our approach, we deployed the proposed coding based approach on top of in-network combiner (*Combine-N-Code*).

Figure 12 shows the normalized *VoC* for all four scenarios using the *Grep* benchmark for both topologies Top-1 and Top-2. Note that the normalized VoC of the proposed coding based approach is the same as $\mu(bisection)$ defined in Section V. The results show that the proposed coding based

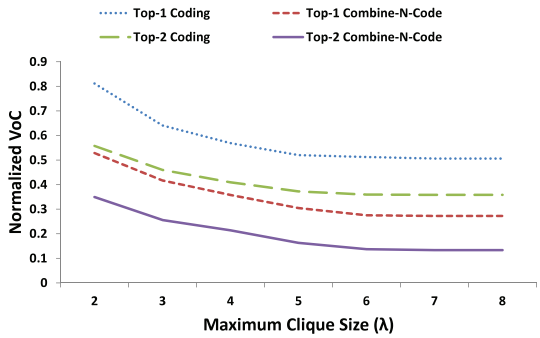

Fig. 14. Normalized *VoC* for different values of λ (maximum clique size) used in Algorithm 2.

approach, compared to vanilla shuffle, can reduce the volume of communication by 43% for Top-1, and 59% or Top-2. Moreover, the coding based approach outperforms in-network combiner both for Top-1 and Top-2 by 13% and 38% respectively. Furthermore, deploying *Combine-N-Code* can reduce the volume of communication by a staggering 64% for Top-1, and 79% for Top-2.

Figure 13 shows the normalized *VoC* for all four scenarios using the *Sorting* benchmark for both topologies Top-1 and Top-2. For the sorting benchmark in-network combiner did not reduce volume of communication at all. Whereas the proposed coding based scheme leads to a 37% reduction volume of communication for Top-1, and 62% reduction in volume of communication for Top-2 compared to both vanilla shuffle as well as in-network combiner.

It is interesting to note that the coding based approach reduces more network traffic on Top-2 as compared to Top-1. The reason for better performance for Top-2 is due to the specific placement of the middlebox resulting in more data exchanged through it, and hence giving rise to more coding opportunities. Specifically, for Top-1 56% and 67% of the data exchange happened through the $L2 - Aggregate$ switch (co-located with the middlebox) for *Grep* and *Sorting* respectively. Whereas for Top-2 most of the data exchanged passed through the $L2 - switch$ (co-located with the middlebox) giving rise to more coding opportunities.

The communication overhead associated with side information gathering for the *sampler* for *Grep* and sorting were less than 1% (0.65% and 0.4%).

Figure 14 shows the normalized *VoC* for different values of λ (maximum clique size) used in Algorithm 2. Increasing λ increases the coding advantage (decreases *VoC*). The maximum coding advantage is achieved at $\lambda = 8$ which conforms the discussion in Section VII.

2) Goodput: In this Section we compare the proposed approach with vanilla shuffle for *GP* measured at the receiver service instance. Figure 15 shows that the coding based scheme outperforms vanilla shuffle at all link rate settings. Moreover, the coding benefit increases with the increase in link rates (*GP* for coding based scheme is is 55% higher than vanilla shuffle at 500 Mbps and grows to 76% at 1000 Mbps).

Moreover, we investigate the impact of oversubscription ratio on *GP* for different link rates. The oversubscription ratios were

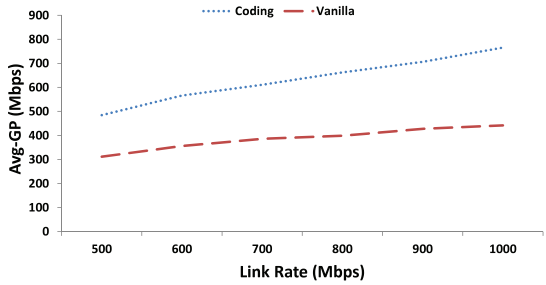

Fig. 15. Goodput versus link rates for sorting benchmark for topology Top-1.

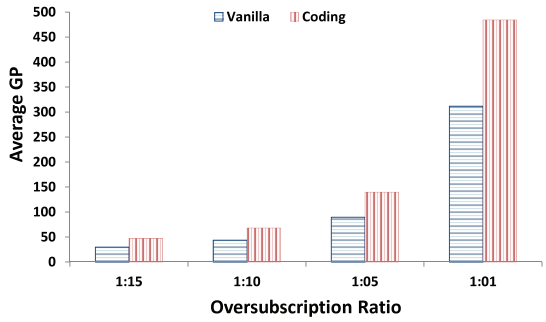

Fig. 16. Goodput for different oversubscription ratios using sorting benchmark for topology Top-1 with link rate at 500 Mbps.

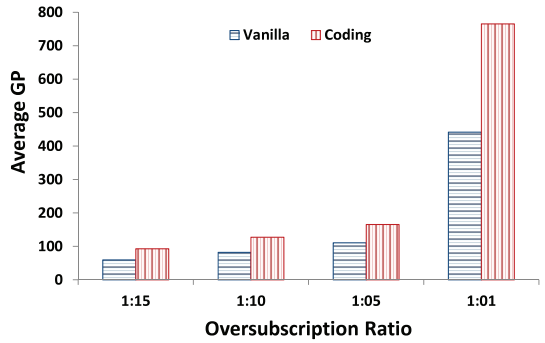

Fig. 17. Goodput for different oversubscription ratios using sorting benchmark for topology Top-1 with link rate at 1000 Mbps.

implemented by generating constant bit rate background TCP tarffic using *iperf* tool. Figure 16 shows the average *GP* for oversubscription ratios of 1, 5, 10, and 15, where the link rate was fixed at 500Mbps. Figure 17 shows a similar plot for link rate of 1000 Mbps. The coding benefit, averaged over oversubscription ratios, is around 56% for 500 Mbps, and 58% for 1000 Mbps.

3) Disk IO Statistics: In this Section we compare proposed approach with vanilla shuffle for two disk I/O related parameters, i.e., *%d-util* and *Qsz* measured at the receiver service instance. Figures 18 and 19 show that the coding based scheme outperforms vanilla shuffle at different link rate settings. Furthermore for both *Qsz* and *%d-util*, the percentage improvement in performance between the proposed scheme and vanilla shuffle peaks to more than 39% at link rate of 1000 Mbps. This trend can be explained by observing that as the link rate becomes higher the disk I/O at the receiver service instances becomes the bottleneck which can be compensated

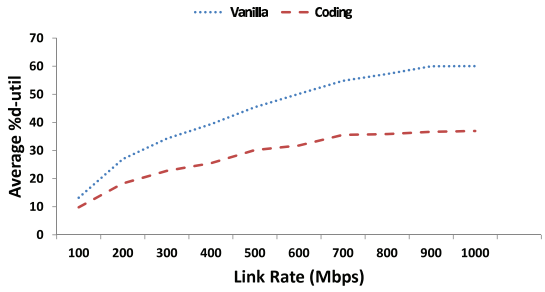

Fig. 18. *%d-util* versus link rates using sorting benchmark for topology Top-1.

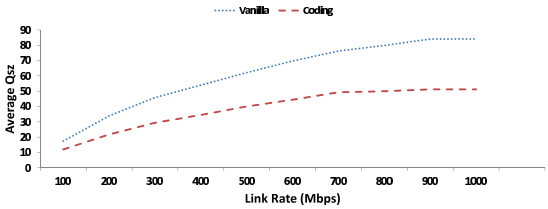

Fig. 19. *Qsz* versus link rates using sorting benchmark for topology Top-1.

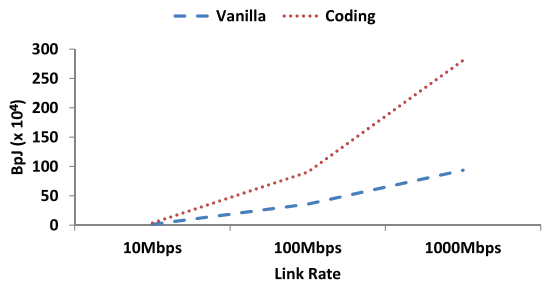

Fig. 20. *BpJ* versus link rates using sorting benchmark for topology Top-1.

by reduction in the volume of communication offered by the coding based scheme.

%d-util and *Qsz* are recorded for the same duration for both vanilla shuffle as well as coding based scheme.

4) Bits-Per-Joule: In this Section we compare *BpJ* for the proposed approach with vanilla shuffle. We investigated possible improvement in energy efficiency by changing Open vSwitch link rates in the testbed and using [61] to calculate *BpJ*.

Figure 20 shows *BpJ* for different link rates. The improvement in *BpJ* of coding based scheme over vanilla shuffle grows significantly with higher link rates (i.e., 2.5, 4.5, and 187.2 more BpJ at link rates of 10, 100, and 1000 Mbps respectively).

In general by choosing lower link rates, favouring lower power (energy per unit time) consumption, the corresponding *GP* also drops as much more time is elapsed in network. We observe that the benefit of choosing lower link rate to improve power efficiency might not be an energy efficient solution for certain scenarios as deduced by the results of Figure 20 that the *BpJ* grows with the increase in link rate.

IX. Conclusion

This paper introduces the novel concept of *spate coding* for reducing the volume of communication, without degrading the rate of information exchange, in big data processing frameworks. We have introduced for the first time, to our knowledge, a network middlebox service that employs *spate coding* and software-defined networking to optimize network traffic. We have not only analyzed the computational complexity of a general schemes for minimizing the volume of communication, but also provided the bounds on maximum advantage and placement strategies. We have also performed a proof-of-concept implementation in real world practical scenarios. Moreover, we performed several experiments to investigate the purposed scheme in a holistic fashion including the parameters capturing both the network and the host nodes. The performance indicators including volume of communication, goodput, bits per Joule, disk utilization, and queue size showed the performance of the proposed system across a spectrum of scenarios and applications.

Appendix A
Spate Coding and Its Relationship With Network Coding and Index Coding

In this section, we describe *spate coding* problem in context of traditional *network coding* and *index coding* problems. We begin by noting that though all these coding problems exploit mixing of the packets, each of these target a different environment that effect the problem characterization and solution space. Index coding problem was first introduced in 1998 [36], and network coding was first introduced in 2000 [62], and it was not until 2015 [63], [64] when a first comprehensive relationship was established between these problems, though some previous work focused on establishing relationship for some specific scenarios (e.g., [65]. We emphasize that *spate coding* problem is different from both the standard network coding and index coding problems [35]–[39], [66]–[68]. Specifically, *spate coding* problem incorporates both network coding and index coding problems as given by Theorems 5, and 6.

We first start by highlighting the relationship between index coding problem and *spate coding* problem. Specifically, *spate coding* problem subsumes index coding problem as given by Theorem 5.

Theorem 5: For each instance of *index coding* problem there exists a corresponding instance of *spate coding* problem.

The proof of Theorem 5 is given in Section B.

Index coding problem is closely related to *spate coding* problem by virtue of incorporating *side information*. However, index coding problem has been defined for the networks communicating over a *broadcast channel*, whereas *spate coding* problem incorporates the networks communicating over a *non-broadcast* channel. *Spate coding* extends and generalizes the concept of index coding problem to *non-broadcast* environments.

Furthermore, an optimal solution for index coding problem is not an optimal solution to *spate coding* problem. We briefly present an example where the optimal solution of index coding problem is not optimal for *spate coding* problem rather it is counterproductive and results in more number of link-level transmissions (even greater than traditional routing). Consider an instance of *spate coding* problem as shown in Figure 21(a).

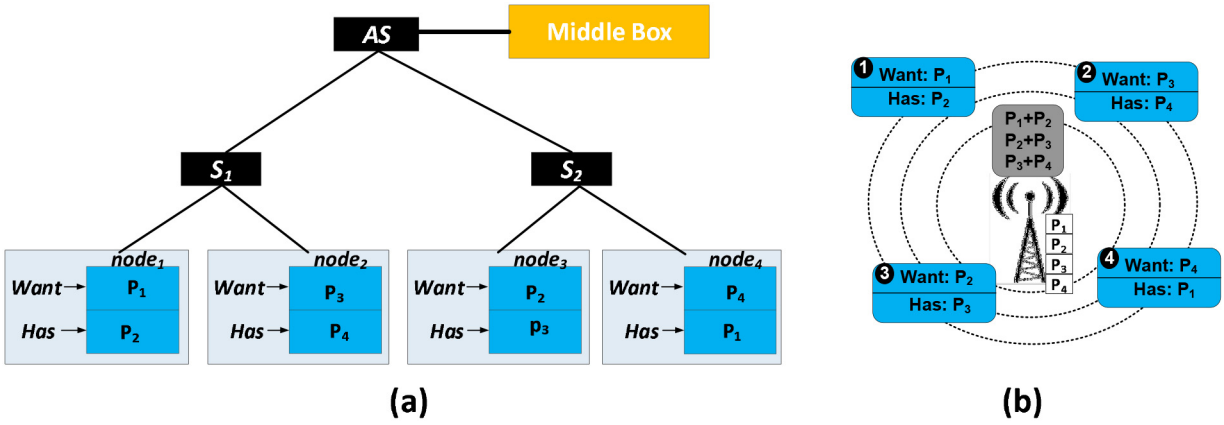

Fig. 21. (a) An instance of *spate coding* problem. (b) A corresponding instance of index coding problem.

A corresponding instance of index coding problem is shown in Figure 21(b). It can be verified that the number of link-level packet transmissions required with traditional routing is 16 to complete the exchange task in Figure 21(a). In contrast if we use the optimal solution for index coding problem to complete the exchange task in Figure 21(a) it requires 26 link-level packet transmissions in total to complete the exchange task (eight link-level packet transmissions for uplink plus eighteen link-level packet transmissions for downlink).

Next, we proceed to briefly describe the relationship between network coding problem and *spate coding* problem. Theorem 6 shows that *spate coding* problem incorporates network coding problem.

Theorem 6: For each instance of *network coding* problem there exists a corresponding instance of *spate coding* problem.

The proof of Theorem 6 is given in Section C.

We note that though network coding and *spate coding* problems are closely related to each other there exists subtle differences. On similarity point, *Spate coding* problem is specific to multiple unicast sessions over an undirected network, and is therefore related to a well studied class of the network coding problem. It has been shown that network coding does not offers any advantage for multiple independent unicast sessions over an undirected network [69], [70]. The assumption of independent upcast sessions does not hold for *spate coding*, and is one of the reasons due to which *spate coding* offers advantage. On dissimilarity point, there are two subtle differences between network coding and *spate coding*. One, for general network coding problem—as per standard problem definition [66], [71]—any intermediate node in the information flow graph can perform encoding of incoming packets, whereas in *spate coding* problem it is not the case rather just a very limited number of nodes in the information flow graph can perform coding (e.g., one node in Example A). This restriction can significantly impact the solution space, and the achievable throughput. Two, *spate coding* problem incorporates *side information*, whereas network coding does not incorporate it. It is interesting to note that though it might be possible to transform some instance of *spate coding* problem into equivalent instances of network coding problem, e.g., by treating side information as additional sources, but in general it is an interesting open problem to show the counterpart of Theorem 6.

APPENDIX B
PROOFS

A. Proof of Theorem 1

We prove the Problem $\mathbb{ES}$ NP-hard by reducing index coding problem into it. It has been already been shown that index coding problem is NP-hard, and NP-hard to approximate [72].

For sake of completeness we start by presenting the definition of index coding problem [39]. An instance of index coding problem is defined by a relay R which contains a set of k packets $X = \{x_1, \cdots, x_k\}$ that are to delivered to a set of m clients $\{c_1, \cdots, c_m\}$ over a *broadcast channel* CHL. Each client c_i has access to some *side information* $H_i \subseteq X$, and requires a set packets $W_i \subseteq X$ from the relay R. The relay R can transmit packets in X or their combinations (encoding). The objective is to find a scheme that requires the minimum number of transmissions from the relay R, and satisfies the requests of all the clients. Let $OPT(IC)$ denote the optimal solution of index coding problem, and $|OPT(IC)|$ denote the minimum number of transmissions from the relay.

Given an instance of index coding problem, we define an instance of the Problem $\mathbb{ES}$ as follows:

Let $|P(AS)|$ denote the number of link-level packet transmissions "made" by the AS in the optimal scheme S. We show that $|OPT(IC)| = |P(AS)|$. We start by analyzing the link-level transmissions in the network $\mathbb{N}$. First note that all the nodes from the set $\{node_1, \cdots, node_k\}$ do not receive any packet during the communication phase of distributed data center application as these do not host any service instance that intend to receive any packet. Furthermore, all the packets in the set U have to pass through the AS; as for each of the packet in U there is a corresponding receiver service instance hosted on one of the nodes $\{node_{k+1}, \cdots, node_T\}$. Second note that for all the packets going through AS destined to a service instance running on any of the nodes $\{node_{k+1}, \cdots, node_T\}$, $OPT(S)$ requires AS to choose the optimal scheme that minimizes the number of link-level packet transmissions on the link CHL. AS can either encode or route the packets as a part of its optimal scheme by utilizing the *side information* (i.e., output of the service instances running on nodes $\{node_{k+1}, \cdots, node_T\}$. It is easier to see that the relay R can use the same schemes as of AS to minimize its number of transmissions which shall be exactly

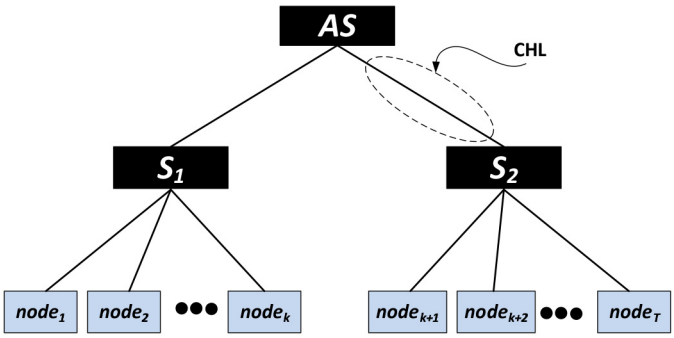

Fig. 22. An instance of the Problem $\mathbb{ES}$.

the same as the link-level packet transmissions by AS on the link CHL in Figure 22.

B. Proof of Theorem 5

The proof follows directly from Section A. Specifically, for each instance of index coding problem a corresponding instance of *spate coding* problem can be constructed using the construction given in Section A.

C. Proof of Theorem 6

The proof follows from Theorem 5, and the existence of on an instance of index coding problem for each instance of network coding problem [63].

D. Proof of Theorem 2

Let ζ denote the number of sender service instances in the network (e.g., number of the mappers in Hadoop MapReduce). Without loss of generality we assume that each sender service instance is required to communicate exactly one packet to its corresponding receiver service instance, if it needs to transmit more than one packet we replace it with multiple sender service instances where each needs to communicate just one packet. We note that d is the maximum number of hops between any sender service instance and its corresponding receiver service instance. Hence, in a non-coding based solution a sender service instance would require at most d link-level packets to communicate its packet to corresponding receiver service instance. As there are ζ sender service instances so a non-coding based solution would require at most $d \cdot \zeta$ link-level packet transmissions. Whereas the coding based solution requires at least ζ link-level packet transmissions to the coding server, one from each of sender service instances before it can even proceed with coding. Hence, $\mu \geq \frac{\zeta}{d \cdot \zeta} = \frac{1}{d}$.

E. Proof of Theorem 3

Directly follows from the fact that a coding based solution can not require more transmissions than a non-coding based solution. The tightness of the bound follows from the observation that in all such instances where receiver service instance do not possess any *side information* a coding based solution can not perform better than non-coding based solution and then $\mu = 1$.

F. Proof of Theorem 4

Without loss of generality we ignore all the sender and receiver service instances that can communicate with each other without passing through the network's bisection, as such instances do not contribute to $\mu(bisection)$. We further assume, without loss of generality, that each sender service instance requires to communicate exactly one packet to its corresponding receiver service instance, if it needs to transmit more than one packets we replace it with multiple sender service instances where each needs to communicate just one packet. Next we note a non-coding based solution would result in two link-level packet transmissions crossing the network's bisection for each of ζ sender service instances, one from sender towards the bisection-switch and other from bisection-switch towards receiver, resulting in a total of 2ζ link-level packet transmissions crossing the network's bisection. Whereas the coding based solution first requires at least ζ link-level packet transmissions crossing the network's bisection to the coding server, one from each of sender service instances. Hence $\mu \geq \frac{\zeta}{2\zeta} = \frac{1}{2}$.

Regarding tightness of the bound, $\mu(bisection) \rightarrow \frac{1}{2}$ for the tree topologies where β sender service instances are located at one side of the bisection-switch (root), and β receiver service instances are located on the other side of the bisection-switch. Moreover, each receiver service instance possesses the demands of all the other receiver service instances as its *side information*. It is easier to see in such a scenario non-coding based solution requires 2β link-level packet transmissions crossing the network's bisection. Whereas coding based solution requires $\beta + 1$ link-level packet transmissions crossing the network's bisection, one from each of β sender service instances to the coding server, and one encoded transmission from the coding server to the receiver service instances. Where the encoded link-level packet transmission consists of bitwise XOR of all the demands of the receiver service instance. Hence in such scenario, $\mu(bisection) = \frac{\beta+1}{2\beta}$, and $\mu(bisection) \rightarrow \frac{1}{2}$ for $\beta \gg 1$.

G. Proof of Corollary 2

Theorem 4 proves if the coding is performed at the network bisection then $\mu(bisection) \geq \frac{1}{2}$. We prove this corollary by presenting an example where $\mu < \frac{1}{2}$ when middlebox for performing coding is not placed at the network bisection, specifically in the example presented in Section B $\mu(bisection) = \frac{1}{4}$ when the coding is not performed at the bisection.

ACKNOWLEDGMENTS

The authors would like to thank Kostas Katrinis with IBM Research.

REFERENCES

[1] Z. Asad, M. Asad Rehman Chaudhry, and D. Malone, "Codhoop: A system for optimizing big data processing," in *Proc. IEEE Int. Syst. Conf. (SysCon)*, 2015, pp. 295–300.

[2] Cisco. (2013). *Cisco Global Cloud Index: Forecast and Methodology, 2012–2017*, White Paper [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf/

[3] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 98–109, 2011.

[4] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 338–347, 2010.

[5] J. G. Koomey, "Worldwide electricity used in data centers," *Environ. Res. Lett.*, vol. 3, no. 3, p. 034008, 2008.

[6] M. Gupta and S. Singh, "Using low-power modes for energy conservation in ethernet LANs," in *Proc. IEEE INFOCOM*, 2007, pp. 2451–2455.

[7] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall, "Reducing network energy consumption via sleeping and rate-adaptation," in *Proc. USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2008, pp. 323–336.

[8] C. Gunaratne, K. Christensen, B. Nordman, and S. Suen, "Reducing the energy consumption of ethernet with adaptive link rate (ALR)," *IEEE Trans. Comput.*, vol. 57, no. 4, pp. 448–461, Apr. 2008.

[9] A. Carrega, S. Singh, R. Bruschi, and R. Bolla, "Traffic merging for energy-efficient datacenter networks," in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst. (SPECTS)*, 2012, pp. 1–5.

[10] C. Gunaratne, K. Christensen, and B. Nordman, "Managing energy consumption costs in desktop PCs and LAN switches with proxying, split TCP connections, and scaling of link speed," *Int. J. Netw. Manage.*, vol. 15, no. 5, pp. 297–310, 2005.

[11] High Scalability. (2012). *Changing Architectures: New Datacenter Networks Will Set Your Code and Data Free* [Online]. Available: http://highscalability.com/blog/2012/9/4/changing-architectures-new-datacenter-networks-will-set-your.html

[12] A. Greenberg *et al.*, "Vl2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.

[13] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a next generation data center architecture: Scalability and commoditization," in *Proc. ACM Workshop Program. Routers Extensible Serv. Tomorrow (PRESTO)*, 2008, pp. 57–62.

[14] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[15] Apache Storm. [Online]. Available: http://storm-project.net/

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41. pp. 59–72, 2007.

[17] Y. Yu *et al.*, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. USENIX Conf. Oper. Syst. Des. Implement. (OSDI)*, 2008, pp. 1–14.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Conf. Netw. Syst. Des. Implement. (NSDI)*, 2011, pp. 323–336.

[19] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM 22nd Symp. Oper. Syst. Principles (SIGOPS)*, 2009, pp. 261–276.

[20] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM EuroSys*, 2010, pp. 265–278.

[21] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, "Transparent and flexible network management for big data processing in the cloud," in *Proc. USENIX HotCloud*, 2013.

[22] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proc. USENIX Conf. Netw. Syst. Des. Implement. (NSDI)*, 2011, pp. 309–322.

[23] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. Conf. Netw. Syst. Des. Implement. (NSDI)*, 2010, p. 19.

[24] N. Farrington *et al.*, "Helios: A hybrid electrical/optical switch architecture for modular data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 339–350, 2011.

[25] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proc. ACM HotSDN*, 2012, pp. 103–108.

[26] M. Chaudhry, Z. Asad, A. Sprintson, and M. Langberg, "On the complementary index coding problem," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, 2011, pp. 244–248.

[27] M. Chaudhry, Z. Asad, A. Sprintson, and M. Langberg, "Finding sparse solutions for the index coding problem," in *Proc. IEEE GLOBECOM*, 2011, pp. 1–5.

[28] A. Curtis, K. Wonho, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, 2011, pp. 1629–1637.

[29] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proc. USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2012, p. 3.

[30] X. Lin, Z. Meng, C. Xu, and M. Wang, "A practical performance model for hadoop mapreduce," in *Proc. IEEE CLUSTER Workshops*, 2012, pp. 231–239.

[31] M. V. Neves, C. A. De Rose, K. Katrinis, and H. Franke, "Pythia: Faster big data in motion through predictive software-defined network optimization at runtime," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2014, pp. 82–90.

[32] D. Perino, M. Varvello, and K. Puttaswamy, "ICN-RE: Redundancy elimination for information-centric networking," in *Proc. ACM SIGCOMM ICN Workshop*, 2012, pp. 91–96.

[33] Cisco. (2011). *Cisco WAAS 4.4.1 Context-Aware DRE, The Adaptive Cache Architecture* [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/routers/wide-area-application-services-waas-software/white_paper_c11-676350.html

[34] Steelhead. *Riverbed* [Online]. Available: http://www.riverbed.com/products/wan-optimization/

[35] R. Ahlswede, N. Cai, S. Li, and R. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.

[36] Y. Birk and T. Kol, "Informed-source coding-on-demand over broadcast channels," in *Proc. IEEE INFOCOM*, 1998, pp. 1257–1264.

[37] Z. Bar-Yossef, Y. Birk, T. S. Jayram, and T. Kol, "Index coding with side information," *IEEE Trans. Inf. Theory*, vol. 57, no. 3, pp. 1479–1494, Mar. 2011.

[38] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in the air: Practical wireless network coding," in *Proc. ACM SIGCOMM*, 2006, pp. 497–510.

[39] M. Chaudhry and A. Sprintson, "Efficient algorithms for index coding," in *Proc. IEEE INFOCOM Workshops*, 2008, pp. 1–4.

[40] S. Sahoo, D. Nikovski, T. Muso, and K. Tsuru, "Electricity theft detection using smart meter data," in *Proc. IEEE Power Energy Soc. Innov. Smart Grid Technol. Conf. (ISGT)*, 2015, pp. 1–5.

[41] ISSDA. *Irish Social Science Data Archive* [Online]. Available: http://www.ucd.ie/issda/

[42] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly, 2012.

[43] R. Lapuh, Y. Zhao, W. Tawbi, J. M. Regan, and D. Head, "System, device, and method for improving communication network reliability using trunk splitting," U.S. Patent 7 173 934, Feb. 6, 2007.

[44] Cisco. (2008). *Data Center Multi-Tier Model Design* [Online]. Available: http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_4.pdf

[45] *Apache Storm.* [Online]. Available: https://storm.incubator.apache.org/documentation/Concepts.html

[46] *Apache Hadoop Package* [Online]. Available: https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/io/package-summary.html

[47] *Hortonworks Data Platform System Administration Guides*, Santa Clara, CA, USA: Hortonworks, 2015.

[48] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.

[49] J. Norris. (2013). *Package org.apache.hadoop.examples.terasort, Apache Hadoop* [Online]. Available: https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[50] Cisco. "Big data in the enterprise—Network design considerations," White Paper, 2011.

[51] W. Yu, Y. Wang, and X. Que, "Design and evaluation of network-levitated merge for hadoop acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 602–611, Mar. 2014.

[52] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 1–12.

[53] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Mapreduce with communication overlap (MARCO)," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 608–620, 2013.
[54] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU Octave Version 3.0.1 Manual: A High-Level Interactive Language for Numerical Computations*. CreateSpace Independent Publishing Platform, 2009, ISBN 1441413006 [Online]. Available: http://www.gnu.org/software/octave/doc/interpreter
[55] redhat. [Online]. Available: http://www.redhat.com/
[56] CentOS. [Online]. Available: https://www.centos.org/
[57] Citrix. [Online]. Available: http://www.citrix.com/products/xenserver/
[58] XenServer. [Online]. Available: http://xenserver.org/partners/developing-products-for-xenserver.html
[59] Open vSwitch. [Online]. Available: http://openvswitch.org/
[60] Y. Yu, P. Kumar, and M. Isard, "Distributed aggregation for data parallel computing," in *Proc. Symp. Oper. Syst. Principles (SOSP)*, 2009, vol. 9, pp. 11–14.
[61] B. Zhang, K. Sabhanatarajan, A. Gordon-Ross, and A. George, "Real-time performance analysis of adaptive link rate," in *Proc. Local Comput. Netw.*, 2008, pp. 282–288.
[62] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.
[63] H. Maleki, V. R. Cadambe, and S. Jafar, "Index coding: An interference alignment perspective," *IEEE Trans. Inf. Theory*, vol. 60, no. 9, pp. 5402–5432, Sep. 2014.
[64] M. Effros, S. El Rouayheb, and M. Langberg, "An equivalence between network coding and index coding," *IEEE Trans. Inf. Theory*, vol. 61, no. 3, pp. 2478–2487, May 2015.
[65] S. El Rouayheb, A. Sprintson, and C. Georghiades, "On the index coding problem and its relation to network coding and matroid theory," *IEEE Trans. Inf. Theory*, vol. 56, no. 7, pp. 3187–3195, Jul. 2010.
[66] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Trans. Netw.*, vol. 11, no. 5, pp. 782–795, Oct. 2003.
[67] B. Hassanabadi, L. Zhang, and S. Valaee, "Index coded repetition-based MAC in vehicular ad-hoc networks," in *Proc. IEEE 6th Consum. Commun. Netw. Conf. (CCNC)*, 2009, pp. 1–6.
[68] S. Sorour and S. Valaee, "An adaptive network coded retransmission scheme for single-hop wireless multicast broadcast services," *IEEE Trans. Netw.*, vol. 19, no. 3, pp. 869–878, Jun. 2011.
[69] Z. Li and B. Li, "Network coding: The case of multiple unicast sessions," in *Proc. Allerton Conf. Commun.*, 2004, vol. 16.
[70] T. Xiahou, Z. Li, C. Wu, and J. Huang, "A geometric perspective to multiple-unicast network coding," *IEEE Trans. Inf. Theory*, vol. 60, no. 5, pp. 2884–2895, May 2014.
[71] T. Ho and D. Lun, *Network Coding: An Introduction*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[72] M. Langberg and A. Sprintson, "On the hardness of approximating the network coding capacity," *IEEE Trans. Inf. Theory*, vol. 57, no. 2, pp. 1008–1014, Feb. 2011.

Zakia Asad (S'13) is currently pursuing the Ph.D. degree in electrical and computer engineering at the University of Toronto, Toronto, ON, Canada. He was the recipient of the Schlumberger Fellowship, University of Toronto Scholarship, Texas A&M ECE Scholarship, a PITB Fellowship, and the Professor KU Medal.

Mohammad Asad Rehman Chaudhry (M'10) received the Ph.D. degree in electrical and computer engineering from Texas A&M University, College Station, TX, USA.

He is an Executive Director with Soptimizer, Toronto, ON, Canada. He has held several industrial and academic positions including a member of High Performance and Exascale Systems Team with IBM Research, a Research Fellow with Hamilton Institute, University of Toronto, a Network Scientist with RCUH for DARPA System F6, an Assistant Director of Huawei-University of Engineering and Technology Joint Telecom Center, as well as a Faculty Member of the Electrical Engineering with the University of Calgary, and the University of Engineering and Technology.

Dr. Chaudhry is the Chair of IEEE Working Group P1916.1, where he leads development of the standards for Software-Defined Networking and Network Function Virtualization Performance. He was the Vice Chair for Performance at the IEEE Standards Study Group for Security, Reliability, and Performance for Software Defined and Virtualized Ecosystems. He is the Chair of the IEEE Research Group on Standards related to SLAs for Virtualized Environments. He is a General Chair of the IEEE GlobalSIP 2015 Symposium on Signal and Information Processing for Software-Defined Ecosystems and Green Computing. He is also a Technical Chair of the IEEE GLOBECOM 2015 Workshop on Green Standardizations for ICT and Relevant Technologies. He is an editor for IEEE Technical Committee on Green Communications and Computing's Green Forum newsletter. He received the Fulbright Fellowship, Presidential Scholarship, Texas A&M ECE Scholarship, and Texas A&M Class Star Award for academics.

David Malone received the B.A.(mod.), M.Sc. and Ph.D. degrees in mathematics from Trinity College Dublin, Dublin, Ireland. During his time as a postgraduate, he became a member of the FreeBSD Development Team. He is currently a member of the Hamilton Institute and the Department of Mathematics and Statistics, Maynooth University. His research interests include mathematical modeling and measurement of WiFi, PLC, and password use. He also works on IPv6 and systems administration. He is a coauthor of O'Reilly's "IPv6 Network Administration."