Virtualized Network Coding Functions on The Internet

Linquan Zhang*, Shangqi Lai†, Chuan Wu†, Zongpeng Li*, Chuanxiong Guo‡
* University of Calgary, {linqzhan,zongpeng}@ucalgary.ca
† The University of Hong Kong, aquas@connect.hku.hk, cwu@cs.hku.hk
‡ Microsoft Research, chguo@microsoft.com

***Abstract*—Network coding is a fundamental tool that enables higher network capacity and lower complexity in routing algorithms, by encouraging the mixing of information flows in the middle of a network. Implementing network coding in the core Internet is subject to practical concerns, since Internet routers are often overwhelmed by packet forwarding tasks, leaving little processing capacity for coding operations. Inspired by the recent paradigm of network function virtualization, we propose implementing network coding as a new network function, and deploying such coding functions in geo-distributed cloud data centers, to practically enable network coding on the Internet. We target multicast sessions (including unicast flows as special cases), strategically deploy relay nodes (network coding functions) in selected data centers between senders and receivers, and embrace high bandwidth efficiency brought by network coding with dynamic coding function deployment. We design and implement the network coding function on typical virtual machines, featuring efficient packet processing. We propose an efficient algorithm for coding function deployment, scaling in and out, in the presence of system dynamics. Real-world implementation on Amazon EC2 and Linode demonstrates significant throughput improvement and higher robustness of multicast via coding functions as well as efficiency of the dynamic deployment and scaling algorithm.**

I. Introduction

Network coding is a simple yet elegant technique that encourages information mixing as data flows traverse a network. Departing from the traditional store-and-forward paradigm, network coding represents a general routing model where nodes (*e.g.*, routers) in the network can both *forward* and *encode* packets. Extensive theoretical studies on network coding have identified a number of advantages, *e.g.*, higher end-to-end throughput, lower transmission cost, and achieving network capacity with low-complexity algorithms [1], [2], [3], [4]. Because of practical constraints, routers on today's Internet mostly forward or drop received packets only, with no capacity equipped for computation-intensive tasks such as packet encoding and decoding. Real-world implementation of network coding has been confined to the overlay layer, *e.g.*, in a peer-to-peer (P2P) network [5], [6], [7]. Due mainly to the unreliability of peers, P2P-based network coding has not seen mainstream deployment in the Internet.

This work was supported in part by grants from Hong Kong Research Grants Council under the contracts HKU 718513, 17204715, 17225516, 717812 and C7036-15G (CRF), grants from the National Natural Science Foundation of China under the contracts NSFC 61571335 and NSFC 61628209, and HKU URC Matching Funding.

Latest developments of the network function virtualization (NFV) paradigm shed light on practical, reliable deployment of network coding. Leveraging software implementation that runs on virtualization-based platforms and industry-standard hardware, NFV aims to enable great flexibility and cost effectiveness in deployment and maintenance of network functions, which used to be realized on dedicated hardware [8]. Without relying on routers equipped with coding capacity, we advocate implementing network coding as a new type of virtual network function (VNF), and deploying coding functions in geo-distributed cloud data centers, to practically enable network coding for bandwidth-efficient flow transmission on the Internet. Coding function deployment is in the charge of service providers managing large-scale flow transmissions, *e.g.*, content distribution service providers hosting unicast and multicast sessions, video conferencing service providers with conference streaming sessions. Network coding functions can be used for bandwidth efficiency in their systems, with low-cost, dynamic deployment of the relay and coding nodes.

As compared to P2P-based and other overlay-based network coding systems, implementing network coding as a VNF embraces great flexibility and cost effectiveness while guaranteeing reliability and performance predictability, thanks to the central control of deployment by a service provider and the application of NFV paradigm. In addition, VNFs (VMs) deployed in cloud platforms are much more reliable than peers in a P2P system [9]. As compared to classic client-server service (*e.g.*, CDN) based on direct source to destination flows, adding relay nodes may circumvent bandwidth bottlenecks and long-delay links; the network coding function improves throughput in cases of stored data downloading, and achieves a target throughput with less bandwidth consumption in cases of real-time data streaming. The flexibility of the NFV paradigm enables dynamic, minimal-effort deployment of necessary coding nodes according to traffic demand, for cost effectiveness at the service provider.

This work aims to virtualize network coding as a type of network function, towards full-fledged implementation of network coding on the Internet, achieving its bandwidth saving potential with minimal modification in network infrastructures. We target multicast from a sender to multiple receivers (subsuming unicast as a special case) as our use cases. As IP multicast is largely unavailable in today's Internet, NFV-based multicast is an effective alternative to support applications such as video conferencing, large file sharing, and on-demand multimedia streaming. Compared with existing application layer multicast systems, *e.g.*, SALM [10], such solution benefits from network coding as well as the new NFV paradigm, enabling higher throughput and great flexibility. Towards such a goal, we address the following design challenges.

First, implementing an *efficient* virtual network coding function is critical for system performance, given that packets are to be coded and forwarded at near line speed. We implement well-optimized virtual machines (VMs) tailored for network coding. As compared to using physical machines, elastic VM provisioning enables agile scaling in and out of VNFs when needed. We exploit the state-of-the-art *data plane development kit (DPDK)* [11] architecture to allow network coding functions to transmit/receive packets directly via network interface cards (NICs), for high coding throughput and low coding latency. For transparency to applications, network coding functions are introduced as a layer between transport layer and application layer in the Internet protocol suite. UDP is selected as the underlying transport protocol for the network coding function. We tune the block/generation sizes and buffer size, to minimize extra delay introduced by coding. A control plane working with network coding functions is carefully designed, to facilitate the following management framework.

Second, an efficient management framework is necessary, helping the service provider route multicast flows to network functions, and decide where to deploy the network coding functions and adjust the deployment over time for best bandwidth efficiency with end-to-end delays. We propose a scaling algorithm, running on a global controller, for dynamic coding function deployment, multicast routing and efficiently handling system dynamics (change of available bandwidths and delays between data centers, addition and termination of multicast sessions, arrival and departure of receivers). The design is rooted in an optimal routing and placement problem, formulated with the help of *conceptual flows* [4] for an optimal tradeoff between throughput and cost in VNF deployment. The dynamic deployment decisions from our algorithm design are practiced through scaling in/out VNFs as needed, which represents a major benefit of using NFV.

A prototype NFV-based network coding suite is designed, implemented, and deployed across geo-distributed cloud comprising of Amazon EC2 and Linode data centers. Real-world experiments demonstrate throughput improvement and higher robustness of multicast via coding functions, as compared to direct sender-to-receiver transmissions and routing-only solutions, at the cost of moderate delay increments (0.9% $\sim$ 1.5%). The experiments also verify that the scaling algorithm can efficiently deploy VNFs to cater for new demand, and recycle the resource after the demand departs.

The rest of the paper is organized as follows. We review related work in Sec. II, and discuss system design and implementation details in Sec. III. The optimization problem for coding function deployment and multicast routing, and the efficient dynamic scaling algorithm are presented in Sec. IV. Experiment results are in Sec. V. Sec. VI concludes the paper.

II. Related Work

A significant body of research has been conducted on multicast in the past few decades. Network layer multicast [12], [13] was proposed and standardized in early 1990s. Yet most parts of the Internet still lack native multicast capability as most commercial ISPs do not widely deploy such a support in their network infrastructure [10]. As a result, efforts [10], [14], [15], [16] have been devoted to application-layer multicast protocol design, which implements multicast forwarding functionality without the need to change the infrastructure. Compared with the existing application-layer multicast solutions, our work focuses on multicast with network coding, while practically implementing network coding using the emerging NFV paradigm, which offer higher throughput and great flexibility.

Network coding has been extensively studied since its proposal at the beginning of the century [1]. Li *et al.* [4] introduce conceptual flows, and use it to analyze how to efficiently compute and achieve maximum multicast rates using network coding. Li *et al.* [17] design a multicast protocol for wireless networks using network coding, focusing on reliability and energy efficiency. Gkantsidis *et al.* [6] design a network coding scheme for P2P content distribution. Chen *et al.* [18] propose a P2P overlay conferencing system based on network coding. Yet the unreliable nature of peers hinders the overall performance. Airlift [19] is a recent cloud service for video conference using network coding, and is observed to outperform P2P based implementations. However, Airlift considers only static scenarios, and fails to scale in/out dynamically in accordance with system dynamics.

Considerable efforts on NFV have been devoted to bridging the performance gap between the dedicated hardware and virtualized network functions. Martins *et al.* [20] propose ClickOS, a high performance, virtualized software middlebox framework. ClickOS requires to modify the underlying hypervisor, making ClickOS-type VNFs unsupported by current cloud platforms that include EC2, Azure and Linode. NetVM [21] enables virtualized network functions to operate at near line speed on top of DPDK. Similar to ClickOS, it needs to modify the hypervisor, which rules out NetVM in our design. Inspired by NetVM, our network coding VNF design leverages DPDK, using poll mode drivers to access NICs for high performance. Different from NetVM, our implementation can be deployed in public cloud platforms.

Along the management of NFV systems, Sherry *et al.* [22] advocate that with the advent of cloud computing, middleboxes in enterprise networks can be replaced by cloud services to enable cost reduction and system elasticity. They propose a service for outsourcing middlebox processing to clouds. Elias *et al.* [23] design an orchestration mechanism to control and reduce resource congestion of the infrastructure in NFV. Gember *et al.* [24] design Stratos, an orchestration layer that offers efficient VNF placement and dynamic scaling algorithm. Yet the above frameworks focus on virtual middleboxes where traffic is simply routed among VNFs. Our network coding function brings unique challenges as it needs to encode/decode packets, making those frameworks not applicable.

Khasnabish *et al.* [25] discuss the impact of NFV and SDN on network coding, as well as the possibility of integration of network coding in various network stack layers. Szabo *et al.* [26], [27] advocate network coding as a service. Yet their prototypes are rather simple, focusing on a single unicast session only. Furthermore, their design cannot be deployed in geo-distributed cloud data centers, while our implementation, on the contrary, aims to utilize geo-distributed cloud data centers to practically enable network coding on the Internet.

III. System Design

As shown in Fig. 1, we consider multicast sessions owned by a service provider, with senders (sources) and receivers (destinations) distributed in different geographic locations. A number of cloud data centers reside across these regions, and the service provider deploys network coding functions in data centers to assist its multicast service. Each network coding function is deployed on one VM in a selected data center.

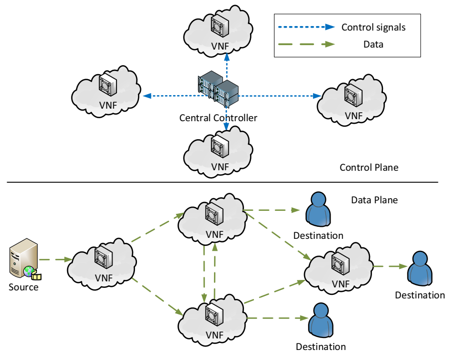

Fig. 1. An example multicast service enabled by network coding VNF: one source and three destinations; control plane and data plane.

The system consists of two planes: the control plane decides where to encode flows, and is responsible for VNF management and multicast routing; the data plane is responsible for receiving and transmitting data and performing data encoding/decoding.

A. The Control Plane

A daemon program runs on each network coding node. A central controller at a high-performance server or VM communicates with the daemons for implementing control-plane functions. As a key component of the network coding NFV system, the controller needs to be fault tolerant. Fault tolerance in NFV is studied in a series of recent work [28], [29], and is orthogonal to this work. The controller launches network coding VNFs in geo-distributed data centers on demand. In each new coding node, daemons start along with initial settings, *e.g.*, IDs of sessions whose flows it is processing, VNF roles (encoder or decoder) associated with different sessions, and UDP ports used for the coding VNF. In the presence of system dynamics, the controller adjusts coding function deployment on the fly, *i.e.*, updating the forwarding tables, terminating existing coding functions and launching new ones. Daemons need to be informed of: start/end of a network coding session, forwarding table updates, VNF initial settings and VNF (VM) shutdown. The signals below are designed to carry these messages from the controller to the VNFs: `NC_START`: starts network coding enabled transmission instead of normal data transmission; `NC_VNF_START`: indicates the number of new VNFs, and initializes these VNFs (VMs) in the corresponding data center; `NC_VNF_END`: informs a VNF that it is no longer used, and shuts it down in τ time; `NC_FORWARD_TAB`: updates the forwarding table; `NC_SETTINGS`: informs the VNF roles, session IDs, UDP ports, generation/block sizes.

The control plane modules are illustrated in Fig. 2. The controller computes a coding deployment and multicast routing scheme by a dynamic algorithm, to be introduced in Sec. IV, specifying the deployment locations and the number of coding functions to deploy in each location. The controller then sends `NC_SETTINGS` to source(s) to inform initial settings, and then further informs next-hop's address(es) via `NC_FORWARD_TAB`. The controller also sends `NC_VNF_START` to itself to start the computed number of VNFs in the data centers decided by the dynamic algorithm. Starting new VNFs (VMs) is implemented by APIs provided by cloud providers, *e.g.*, Linode APIs and EC2 CLI/AMI. The controller then informs the newly started VNFs about their roles, session and port information via `NC_SETTINGS`, and the forwarding table via `NC_FORWARD_TAB`. When a new VNF is ready, the controller sends `NC_START` to inform the daemon on the VNF to start the network coding function.

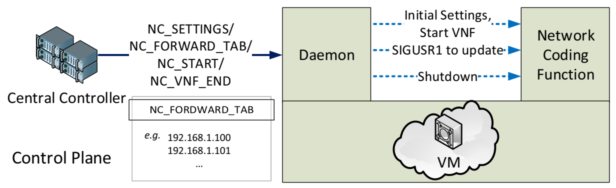

Fig. 2. An illustration of the control plane modules for coding VNFs.

The forwarding table is a text file, recording the next hops' IP addresses for each relevant multicast session the coding function belongs to. After a daemon receives the new forwarding table file, it sends `SIGUSR1`, a user-defined signal in Linux, to temporarily pause its coding function, inform the coding function of the new forwarding table, and then resume the function. After loading the new forwarding table, the network coding function resumes.

Each VNF creates a UDP socket listening at a designated port, and checks if a packet has the network coding protocol header (to be discussed next). Encoders extract the payload of the packets, perform encoding operations, and send the encoded packets out via NIC to next-hop VNFs. When decoder VNFs receive encoded packets, they execute decoding operations and forward the recovered payload to the destinations.

When the network coding topology evolves over time, *e.g.*, due to changes of bandwidth and delay, addition and termination of multicast sessions, arrival and departure of receivers, the controller updates the VNFs' forwarding tables using `NC_FORWARD_TAB`. In case of VNF population changes, `NC_VNF_START` and `NC_VNF_END` are employed to launch and shut down VNFs (VMs), respectively. After a daemon receives a `NC_VNF_END` signal, it shuts down its VNF (VM) in a threshold time τ, instead of immediately, for potential reuse of the VNF if traffic load increases within τ. The idle VNF is shut down after τ for saving operational cost. Such VNF reuse helps mitigate the overhead of launching new VNFs.

B. The Data Plane

1) Network Coding Details: **Generation and Block**. In each multicast session, source data are divided into a number of *generations*, each assigned with a session-wide unique generation number. Within a generation, the data is further divided into *blocks*. Network coding happens within each generation. An encoded block is a linear combination of blocks within a generation. Fig. 3 shows the relation between the original data, generation, blocks and encoded block.

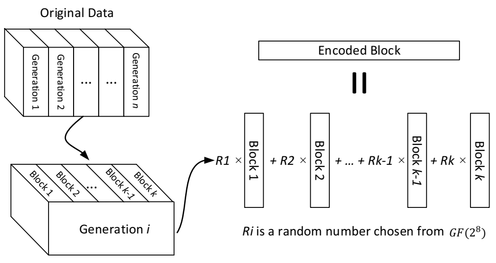

Fig. 3. An illustration of the generation, blocks and encoded blocks.

Randomized Network Coding. We apply randomized network coding (RNC), a powerful coding scheme that guarantees the receivers can successfully recover the original data with a high probability [30]. In RNC, each encoded block is computed by linear combination of blocks within a generation, and the coefficients in the combination are random numbers in a Galois field. We follow the practice in the literature and choose the field $GF(2^8)$, which was observed to enable the maximum throughput among all field sizes [2], [19]. We implement coding by using an open source network coding library: Kodo [31], which offers high performance RNC.

Network Coding over UDP. In the Internet protocol stack, the network coding layer is introduced between the application layer and the transport layer. A network coding (NC) header is designed to carry information related to the network coding scheme, including session ID, generation ID, and encoding coefficients, with a total of 8 bytes plus the length of coefficients, which depends on the number of blocks in each generation. We assume that the system uses the same generation size (number of bytes in a generation) and block size (number of bytes in a block) across all sessions. The generation and block sizes are sent to each network coding function at its initialization stage.

Our network coding function uses UDP as the underlying transport protocol. The TCP retransmission mechanism makes TCP not suitable for our network coding function as our system is not concerned with out-of-order packets or the loss of a single encoded packet. The data can be successfully recovered as long as sufficient number of packets are received. The retransmission mechanism does not help the network coding function, even worse, could deteriorate the performance as the sending rate is reduced when retransmission is triggered.

Block and Generation Sizes. We set the block size to 1460 bytes, which together with the NC header (12 bytes, with 4 blocks in each generation), the UDP header (8 bytes) and the IP header (20 bytes) make up the MTU size (1500 bytes) of NICs in EC2. Such a setting prevents NC packet fragmentation. As for the generation size, if there are many blocks in one generation, the encoding and decoding complexity is high, and the delay for collecting enough encoded blocks for decoding increases, especially in cases of packet losses. If there are very few blocks in a generation, a larger field size is needed for controlling the chance of linear dependency among encoded blocks, which in turn increases the coding complexity [31]. We conduct a comparison study of generation sizes for multicast in a *butterfly* topology, as shown Fig. 6 in Sec. V. Fig. 4 shows that the throughput reaches the maximum when each generation contains four blocks, and plunges when the number of packets is over 16. As a result, we allow each generation to contain 4 blocks in our system, in pursue of high throughput and low latency.

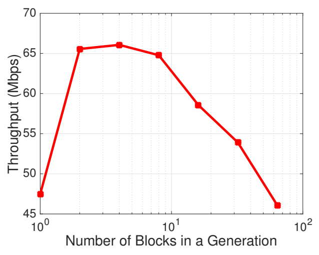

Fig. 4. A comparison among generation sizes; each block = 1460 bytes.

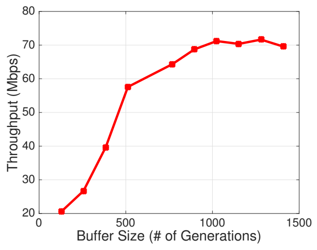

Fig. 5. A comparison among various sizes of buffer.

2) Virtual Network Coding Function: Towards high data processing speed, the network coding function (*i.e.*, the VNF) is designed with the following highlights.

DPDK polling for retrieving packets from NIC. Traditional interrupt based mechanisms, *e.g.*, netfilter [32], interrupt the packets, extract the payload, and pass it to an encoding process in the user mode. Such a mechanism is not suitable for high performance packet processing due to its costly context switching and need of thousands of CPU cycles. The efficiency deteriorates when the number of interrupts grows. Intel's DPDK [11] allows applications in user-space to transmit/receive data directly via NICs using the `poll` mode rather than interrupts, reducing unnecessary context switches for low latency and high throughput. We employ DPDK for receiving and transmitting packets, which further provides a gadget named kernel NIC interface (KNI) [11] for utilizing the kernel network stack for easy packet processing on the Internet. We employ KNI to access the kernel network stack, and assign IP addresses to DPDK enabled NICs. DPDK separates NICs from the Linux kernel, and these interfaces under DPDK control cannot obtain an IP address from the system. This would prevent the deployment of our system on the Internet. KNI provides a channel between interface management in the kernel and the DPDK driver, which makes these interfaces accessible again on the Internet. Additionally, it allows socket-fashion programming in user space for further packet processing, which reduces implementation complexity.

Buffer space is needed for storing packets received so far. A newly arriving packet is stored based on its session ID and generation ID for further encoding operations. The system therefore can quickly encode the newly received packets with existing packets from the same session and same generation. We employ a FIFO buffer management strategy that discards the oldest packets once the buffer is full. We measure the impact of buffer size for a multicast session, as shown in Fig. 5. Results suggest that buffer size of 1024 generations is sufficient to guarantee good performance (larger buffer gains little benefit). Thus, we choose 1024 generations as the buffer size for each session in the system.

We implement the network coding function on each coding node in the socket fashion for packet processing. The virtual network coding function decapsulates received UDP packets from NICs, obtains the coded blocks, which are placed into its buffer. The network coding function processes received packets in a pipelined fashion, *i.e.*, an intermediate VNF generates an encoded packet immediately after it receives a packet from the same session and generation, and sends encoded packets out to the next hops specified in the forwarding table according to session IDs. In case of the packet is the first one in its generation received by the VNF, the VNF simply forwards it.

IV. Dynamic Deployment and Scaling of Network Coding Functions

We then design a dynamic algorithm for coding VNF scaling and multicast routing, executed by the controller.

A. Optimization Problem for Coding Topology Generation

We first formulate the optimization problem of decide the coding function deployment locations, the number of coding functions to deploy, and traffic routing among sources, coding functions, and destinations. There are a set of multicast sessions M ($|M| \geq 1$), each assigned a unique session id by the controller. The source in session m is s_m, and the destinations are $d_m^k, k = 1, ..., K_m$. Let V be the set of data centers that the service provider can access for deploying coding functions, and E be the set of directed links between the data centers and between the data centers and sources/destinations. Each destination is capable of decoding; possibly with the help of a coding VNF in a nearby cloud. The delay along a link $e \in E$ is $L(e)$, which may be time varying. The available inbound bandwidth and outbound bandwidth of a VNF in data center v are $B_{in}(v)$ and $B_{out}(v)$, respectively. It is common for data centers to set a bandwidth cap for incoming and outgoing traffic at each VM from/to the Internet, which can be time varying as well according to our measurements shown in Tab. I. A recent study on inter-data center networking [33] also identified the same phenomenon. Assume VMs of the same hardware configuration are used for deploying the coding functions in the same data center, and each coding function in data center v can encode packets at the maximum rate of $C(v)$ (bytes/s). We allow each VNF in the system to encode data for multiple sessions, up to its capacity.

TABLE I
Time varying inbound and outbound bandwidth for one hour in two EC2 data centers in Oregon and California, respectively.

Time (min)	0	10	20	30	40	50
Oregon In/Out (Mbps)	926 /920	918 /938	906 /889	915 /929	915 /914	893 /881
CA In/Out (Mbps)	919 /928	938 /923	883 /909	924 /917	912 /919	876 /901

Feasible paths. Let L_m^{max} be the max tolerable delay from the source to each destination in session m. For example, if the multicast is for live video streaming or video conferencing, then L_m^{max} is small, to ensure real-time playback at destinations; for file downloading or stored video streaming, the delay can be larger. L_m^{max} restricts the flow paths that a multicast session can use, *i.e.*, the coding VNFs that the flow can be relayed through. Given the set of candidate data centers V, we can decide all feasible paths (whose end-to-end delay is no larger than L_m^{max}) between the source and each destination in a multicast session m, by running a modified depth-first-search (DFS) on the graph consisting of sources, destinations, data centers and the links in-between: the DFS continues to search for paths for each pair of source and destination, as long as the path currently obtained has a delay smaller than L_m^{max} and has no cycles. In practice, the number of candidate data centers is usually small, around $5 \sim 20$. Such DFS can quickly compute the feasible paths in topologies of this scale. Let $P_m^k, \forall m, k$, be the resulting path set from source s_m to destination d_m^k. The set includes the direct path from the source to the destination, if the delay on the direct link is below L_m^{max}.

Decisions. Conceptual flows [4], [19] is a classic technique in network coding for formulating routing problems of coded multicast flows. A multicast session of K destinations is deemed as containing K conceptual flows, each from the source to one destination. The actual multicast flow rate on a link is the maximum of the rates of all conceptual flows using this link. Let f_m^k be the conceptual flow to destination d_k in session m, which may contain multiple paths in set P_m^k. Let $f_m^k(p)$ be the conceptual flow rate along the path $p \in P_m^k$. Let $f_m(e)$ denote the actual flow rate of multicast session m along link e. We have

$$f_m(e) = \max_{k \in \{1,...,K_m\}} \sum_{p \in P_m^k : e \in p} f_m^k(p), \tag{1}$$

where $\sum_{p \in P_m^k : e \in p} f_m^k(p)$ is the total rate of conceptual flow f_m^k going through link e. $f_m(e)$ decides the actual routing of multicast session m: if $f_m(e)$ is zero, the multicast flow is not sent along link e; otherwise, the flow is sent along e at the rate specified by $f_m(e)$. Network coding happens at a data center if multiple incoming flows in the same session arrive at it. Let an integer variable x_v be the number of VNFs to deploy in data center v, to handle all incoming flows of multiple sessions. Our optimization problem is to compute VNF deployment in the data centers (x_v's where $x_v = 0$ indicates no coding function at v) and flow routing for each session ($f_m(e)$), given the pre-computed paths based on delay constraint.

Objectives. Let λ_m indicate the end-to-end throughput of session m, *i.e.*, the rate that each destination in session m can receive the data at. Our objectives are to maximize the throughputs of multicast sessions while minimizing the total number of VNFs deployed (which decides cost of the service provider). To pursue two objectives in the same optimization problem, we combine them into one objective function, by a factor $\alpha \geq 0$ that converts the number of VNFs into the same unit as throughput and also gauges the tradeoff between throughput maximization and deployment cost minimization. The optimization problem is formulated as follows:

$$\text{maximize} \quad \sum_{m} \lambda_m - \alpha \sum_{v \in V} x_v \qquad (2)$$

subject to:

$$\lambda_m \leq \sum_{p \in P_m^k} f_m^k(p), \qquad \forall m, k \qquad (2a)$$

$$\sum_{p \in P_m^k : e \in p} f_m^k(p) \leq f_m(e), \qquad \forall m, k, e \qquad (2b)$$

$$\sum_{m \in M} \sum_{\substack{e:e=(u,v), \\ \forall u \in V \cup \{s_{\tilde{m}}\}_{\tilde{m} \in M}}} f_m(e) \leq B_{in}(v) x_v, \qquad \forall v \in V, \qquad (2c)$$

$$\sum_{\substack{e:e=(u,d_m^k), \\ \forall u \in V \cup \{s_m\}}} f_m(e) \leq B_{in}(d_m^k), \qquad \forall m, k \qquad (2c')$$

$$\sum_{m \in M} \sum_{e:e=(u,v), \forall v \in V} f_m(e) \leq B_{out}(u) x_u, \qquad \forall u \in V, \qquad (2d)$$

$$\sum_{e:e=(s_m,v), \forall v \in V} f_m(e) \leq B_{out}(s_m), \qquad \forall m \in M, \qquad (2d')$$

$$\sum_{m \in M} \sum_{\substack{e:e=(u,v), \\ \forall u \in V \cup \{s_{\tilde{m}}\}_{\tilde{m} \in M}}} f_m(e) < C(v) x_v, \qquad \forall v \in V, \qquad (2e)$$

$$f_m^k(p) \geq 0, f_m(e) \geq 0, \qquad \forall m, k, p, e \qquad (2f)$$

$$\lambda_m \geq 0, x_v \in \mathbb{Z}_0^+, \quad \forall v \in V, m \in M. \qquad (2g)$$

Constraint (2a) defines the throughput of multicast session m according to $\lambda_m = \min_k\{$flow rate of conceptual flow $f_m^k\}$, where flow rate of f_m^k is the overall rate of the conceptual flow along all its possible paths $\sum_{p \in P_m^k} f_m^k(p)$. Constraint (2b) is derived based on the relation between conceptual flow and actual flow in Eqn. (1). Constraint (2c) and (2d) enforce the inbound and outbound bandwidth limits of a VM in each data center. In particular, constraint (2c') defines the inbound bandwidth limits for all destinations, and constraint (2d') limits the outbound bandwidth for all sources. Constraint (2e) indicates that the total incoming flow rate to data center v cannot exceed the overall capacity of VNFs deployed.

The optimization problem is expressive in catering for different scenarios. We can set λ_m to a given multicast rate if the rate is fixed for multicast session m (*e.g.*, in case of live streaming), while focusing on finding the most bandwidth efficient routes of the flow to achieve the end-to-end rate while minimizing coding function deployment cost. In the case that the number of VNFs to deploy in each data center is fixed, *i.e.*, $x(v)$ is given, we can set $\alpha = 0$ and find the best routes to maximize throughputs of the multicast sessions. Without loss of generality, we discuss our scaling algorithm in the following for the case of achieving the best tradeoff between throughput maximization and cost minimization, gauged by α.

Optimization (2) is an integer linear program. To solve it efficiently for practical applications, we can relax the integer constraint (2g), and then use standard LP solvers, *e.g.*, `glpk`, to solve the relaxed program optimally and efficiently. We will then round the fractional solutions to nearest integer values. Alternatively, we can apply certain LP solvers, *e.g.*, `cplex`, to directly solve the integer linear program under practical inputs, which can derive an approximate solution in polynomial time.

An optimal solution of (2) contains the number of VNFs and a set of conceptual flows, guiding the flow routing. In the case where only one flow of a session arrives at a data center V, direct forwarding is sufficient and coding is unnecessary. The controller informs the VNFs at data center V to simply forward all packets from that session. After encoding, the encoded packet is forwarded according to its conceptual flow. If there is only one VNF launched in the next hop data center, then the next hop address is the VNF's address. In case of multiple VNFs launched in one data center, we dispatch the incoming packets across these VNFs based on session id and generation id to those VNFs. Packets belonging to the same generation are dispatched to the same VNF instance. The forwarding information is sent to the VNFs at previous hop nodes of all the incoming flows.

B. Dynamic Scaling Algorithm

The initial coding function deployment and multicast routing can be decided by solving (2). Over time, the coding topology needs to be adjusted, upon the following events: i) available inbound/outbound bandwidth increases or drops; ii) delay between two data centers or between a source/destination and a data center increases or decreases; iii) a new multicast session arrives, or an existing multicast session ends; a new receiver joins a session, or an existing receiver leaves a session.

Bandwidth variation. `Iperf3`, a tool to measure bandwidth, is installed on network coding VNFs and periodically executed to obtain the inbound and outbound bandwidth of a single VNF in a data center. Results are sent to the controller for use of the dynamic scaling algorithm. When the bandwidth on a VNF increases, we may take up the extra bandwidth to increase the throughput of sessions that traverse the VNF, or retain the same flow rate. Increasing flow rate may lead to overload of the VNF handling the link. If the VNF is overloaded, we need to deploy additional VNFs in the same data center for

Algorithm 1 Bandwidth Variation

```
1: function BANDWIDTH_VARIATION
2:     for all $v \in V$ and $v$ is used for conceptual flows do
3:         if $\boldsymbol{B}(v)$ changes by more than $\rho_1\%$ and the situation lasts for $\tau_1$ then
4:             update $B_{in}(v)$ and $B_{out}(v)$;
5:     $g$ = solve (2) based on the current deployment and flows except affected data center and flows using bandwidth input $\boldsymbol{B}$;
6:     if $g >$ current objective value then
7:         scale out according to new $\boldsymbol{x}$ via NC_VNF_START;
8:         update the routing table via NC_FORWARD_TAB;
```

coding the extra flow, which may increase the second term of (2). To strike the best tradeoff, we solve (2) again based on the current VNF deployment and flow routing except the affected data centers and multicast flows using the link whose bandwidth has changed. If the new objective value is larger than the old one, we increase flow rates and deploy new VNFs accordingly; otherwise, we retain the existing rates and the same VNF deployment. When the bandwidth on a VNF drops, which triggers packet loss, the throughput of sessions using the VNF may drop if there would be no new VNF deployment. We compute (2) again based on the current VNF deployment and flow routing except the affected data centers and flows. Such computation may provide new routes for the multicast sessions and new VNF deployment in the affected data centers. The algorithm is shown in Alg. 1.

Algorithm 2 Delay Changes

```
1: function DELAY_CHANGES($\boldsymbol{L}$)
2:     for all $e \in E$ and $e$ is used for conceptual flows do
3:         if $L(e)$ changes by more than $\rho_2\%$ and the situation lasts for $\tau_2$ then
4:             update $P_m^k$: remove or add paths
5:     solve (2) based on the existing VNF deployment and unaffected conceptual flows using new path set $P_m^k, \forall k, m$;
6:     scale out according to new $\boldsymbol{x}$ via NC_VNF_START ;
7:     update the routing table via NC_FORWARD_TAB;
```

Delay changes. Delay changes influence path selection in our flow routing. A path becomes infeasible as its overall delay exceeds the maximum tolerable value, and more feasible paths appear if the delays of links drop. `Ping` is periodically executed on the VNFs to detect delay changes on links. Measurements are sent to the controller. After detecting the increase of delay on a link, we check whether the currently used flow paths are affected. If so, the controller eliminates the affected paths from the feasible path sets $P_m^k, \forall m, k$, and solves (2) again to obtain new flow routes for affected sessions and possible additional VNF deployment based on the current VNF deployment. When the delay on a link drops, some path sets $P_m^k, \forall m, k$ may be expanded, which implies that the solution space is expanded. Optimization (2) is solved again using the new path sets based on existing VNF deployment, to compute new routes for affected sessions and possible new deployment of VNFs. The algorithm is summarized in Alg. 2.

Session arrivals and departures. The flow routing and new

Algorithm 3 Session/Receiver Arrivals and Departures

```
1: function SESSION_JOIN($m'$)
2:     run path preselection for $P_k^{m'}$;
3:     solve (2) for the new session only based on current conceptual flows $f_m^k(p)$;
4:     scale out according to new $\boldsymbol{x}$ via NC_VNF_START;
5:     update the routing table for $f_{m'}^k(p)$ via NC_FORWARD_TAB;
6:
7: function RECEIVER_JOIN($t_k^{m'}$)
8:     solve (2) based on VNF deployment and flows of unaffected sessions, to obtain the conceptual flow for $T_k^m$;
9:     scale out according to new $\boldsymbol{x}$ via NC_VNF_START;
10:    update routing table via NC_FORWARD_TAB;
11:
12: function SESSION/RECEIVER_QUIT
13:    $g_1$ = solve (2) based on existing VNF deployment;
14:    $g_2$ = solve (2) based on existing flow rates;
15:    if $g_1 > g_2$ then
16:        increase corresponding flows;
17:    else
18:        shut down VNFs according to $g_2$ via NC_VNF_END;
19:        update the forwarding table via NC_FORWARD_TAB;
```

coding function deployment for the new session are attained by solving (2), for the new sessions only, exploiting any surplus capacity of existing VNFs. The controller accordingly deploys new VNFs and routes the flow. When a session is terminated, we can either increase the existing flow rates to use any surplus VNF capacities or reduce the number of VNFs for cost saving. The controller recomputes (2) twice, once to derive the throughput increase based on the existing VNF deployment and the second time to decide the VNFs to retain based on the existing flow rates, and compares the objective values. If the former leads to a larger objective value, existing flows will maximally use the freed capacities; otherwise, some VNFs will be shut down without affecting the existing flow rates. Alg. 3 shows the algorithm for this case.

Receiver arrivals and departures. When a new receiver joins a multicast session or when an existing receiver departs, optimization (2) is computed again using the new receiver set based on existing VNF deployment and flows of unaffected sessions. Flows may be rerouted. New VNFs may be launched (new receiver arrival case) and some VNFs can be shut down (receiver departure case). The algorithm is shown in Alg. 3.

Discussions. Upon session changes and receiver changes, VNFs are adjusted immediately. In cases of bandwidth/delay variation, the controller instructs flow rerouting (by updating forwarding table of affected VNFs), as well as VNF launch or shutdown only when the change is larger than a threshold percentage, *e.g.*, $\rho_1\%$ and $\rho_2\%$, and such phenomenon has lasted for a threshold time, *e.g.*, τ_1 and τ_2, to avoid unnecessary scaling in cases of brief spikes of bandwidth/delay. The threshold values depend on VNF launch/termination cost as well as the applications the framework supports. During the adjustment, some VNFs may become under-utilized. If such a situation lasts for a threshold time, the controller re-steers traffic among the under-utilized VNFs in the same data center, and shuts down redundant VNFs. We perform incremental update of the coding topology in all cases of system dynamics, instead of solving the optimization completely anew, to minimize overhead of VNF adjustment and flow migration.

V. Experimental Evaluations

A. Prototype Implementation

We implement a prototype system for evaluating our design and algorithms. A file transmission application is built upon the system for driving the evaluation. VMs in six data centers, including three Amazon EC2 data centers located in California, Oregon and Virginia, and three Linode data centers in Texas, Georgia and New Jersey are rented for deploying coding functions, and acting as sources and destinations.

All VMs in Amazon EC2 are `C3.xlarge` instances, configured with four Intel Xeon E5-2680 v2 cores, 1000 Mbps virtualized NIC and 7.5 GB RAM. EC2 Enhanced Networking [34] is also enabled in these instances, such that SR-IOV is used. SR-IOV [35] is a high performance I/O pass-through system, allowing the NIC to be partitioned into several "Virtual NICs", guaranteeing higher performance, lower latency and lower jitter. These NICs work as general devices in VMs. Our system is built upon these virtual devices for high networking performance. The instances from Linode are configured with one Intel Xeon E5-2680 v2 core, 1GB RAM, 40 Gbps and 125 Mbps for incoming and outgoing traffic, respectively. A server located at the University of Hong Kong is used as the controller, equipped with Intel Xeon CPU E5-2650, 64GB RAM and an Emulex Corporation OneConnect 10Gbps NIC. All VMs are installed with Ubuntu Server 14.04 (v3.19) and DPDK-16.04. DPDK KNI is loaded to create a bridge between the Linux network stack and the DPDK poll mode driver.

Our evaluation aims to verify the following: (i) our network coding VNF can achieve high throughput close to line speed; (ii) the additional latency introduced by network coding is small; (iii) network coding can improve the robustness of flow transmission on the Internet; (iv) our dynamic scaling algorithm can ensure the best tradeoff between throughput and VNF deployment cost at all times. Our experiments include two parts: (1) we examine the performance of our network coding VNF in detail on a fixed butterfly topology in Fig. 6, to test the throughput and coding delay of the VNF, as well as robustness of flow transmission; (2) we evaluate the system in realistic scenarios with multiple multicast sessions to demonstrate the performance of the dynamic scaling algorithm.

B. Evaluation on Butterfly Topology

The butterfly network is a classic example demonstrating benefits of network coding [1]. We deploy VMs in California, Oregon, Virginia and Texas to construct the butterfly network. Total link capacity of all outgoing links of a VNF is bounded by its outbound bandwidth, while total link capacity of all incoming links of the VNF is bounded by its inbound bandwidth. To emulate the bottleneck between T and V_2 in the *butterfly* topology, we restrict the link capacities to the values labelled on the links of Fig. 6 using `netem`. We first test the performance of our network coding VNF using one multicast session where two receivers retrieve a 1000 MB file from one source, with four data centers in-between, each hosting one coding function. The multicast topology and locations of the source, destinations and data centers are given in Fig. 6 (O for Oregon, C for California, T for Texas and V for Virginia).

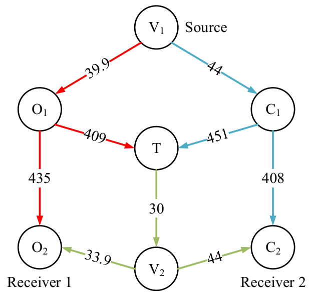

Fig. 6. Butterfly network with VNFs and link bandwidth (Mbps) labelled.

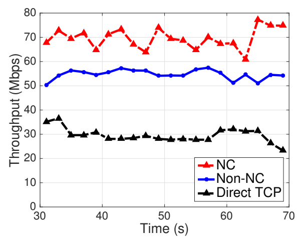

Fig. 7. Throughput comparison in the butterfly topology.

1) Throughput: The throughput of the multicast session is the minimum of the end-to-end rates achieved by the two receivers. We can compute the theoretical maximal throughput of the multicast session using the Ford-Fulkerson algorithm [36], which is 69.9 Mbps. We evaluate the actual multicast throughput with and without coding functions enabled.

In Fig. 7, we observe that rerouting the flows through a number of intermediate data centers leads to higher throughput as compared to the direct connections due to additional bandwidth. Enabling network coding further pushes the throughput approaching the theoretical maximum. It verifies that network coding as VNFs can improve utilization of network resource to increase throughput, in cases that direct connections do not provide good bandwidth. Network coding leads to a throughput close to the theoretical maximum, despite the complicated encoding/decoding processes.

2) Delay: Network coding introduces extra delay to end-to-end flow transmission due to coding computation and packet synchronization (the decoder has to collect a sufficient number of packets to recover a generation). We examine the latency introduced by network coding by measuring the average round-trip delay between source and destinations. End-to-end delays of the following paths are evaluated: (1) from the source in Virginia (V_1) directly to each of the receivers in Oregon and California, respectively; (2) from the source to receiver 1 via the relay nodes; (3) from the source to receiver 2 via the relay nodes. For (1), we measure the round-trip time using standard ping tests with the same packet size as that of our coded packets. For (2) and (3), we measure the round-trip delay when network coding is in place and not (in the latter case, intermediate data centers directly forward received packets), as the time from when the first generation is completely sent out from the source to the time the acknowledge is received back from the receiver (we allow each receiver to send an acknowledge directly back to the source once it has successfully received the (decoded) first generation).

We measure the delays along different paths for multiple

TABLE II
DELAY COMPARISON.

	Receiver O_2 (ms)			Receiver C_2 (ms)		
	min	max	average	min	max	average
Direct path	90.85	90.93	90.879	77.01	77.04	77.03
Relayed path w. coding	167.59	169.75	168.80	167.49	168.92	168.22
Relayed path w/o coding	165.95	168.08	167.27	165.61	168.68	166.46

times, and obtain the minimum, maximum and average round-trip times, as shown in Tab. II. Compared with direct ping test, relayed paths lead to longer round-trip times (but higher throughput). Along the relayed paths, network coding only introduces 0.9% to 1.5% additional delay, which is minor. For example, the average delay increases from $167.27ms$ to $168.80ms$ for path (2), from $166.46ms$ to $168.22ms$ for path (3) when network coding functions are enabled.

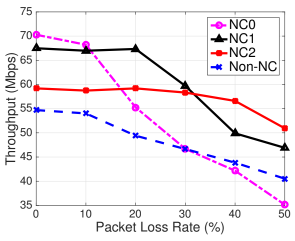

Fig. 8. Throughput comparison at different uniform drop rates.

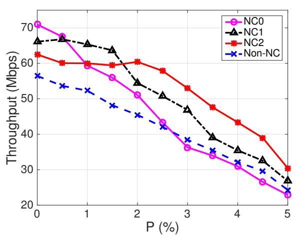

Fig. 9. Throughput comparison at different burst drop rates.

3) *Robustness:* We next examine the impact of packet loss in the links. We emulate i.i.d. random packet loss and burst packet loss in the link between T and V_2 using `netem`. In Fig. 8, we emulate the uniform packet loss rate ranging from 0% to 50%. In Fig. 9, we emulate the burst packet loss, where the loss rate of the n-th packet is $P_n = 25\% \times P_{n-1} + P$, $P_0 = 0$ and P ranges from 0% to 5%. In both packet loss models, we evaluate the impact of packet loss in the following cases: (1) NC0: there is no redundant coded packet produced at each coding node for each generation, *i.e.*, 4 coded packets are produced per generation (since each generation contains 4 blocks); (2) NC1: one extra coded packet is produced per generation; (3) NC2: two extra coded packets per generation; (4) Non-NC: intermediate nodes do forwarding only.

In both figures, we observe the same trend. The throughput of NC0 drops sharply when the packet loss rate increases, since a receiver has to wait for retransmissions to collect all 4 packets for decoding a generation. For the same reason, even Non-NC works better than NC0 at high packet loss rates. The robustness of the system is improved as extra coded packets are added, *e.g.*, the cases of NC1 and NC2. The receivers can recover the original generation, as long as they receive four linearly independent packets rather than all encoded packet for the generation. As compared to Non-NC, NC1 and NC2 can retain a relatively high throughput when there are packet losses. On the other hand, redundancy wastes bandwidth in case of low loss rate, as we can see that the throughput of NC1 is lower than that of NC0 when packet loss rate is close

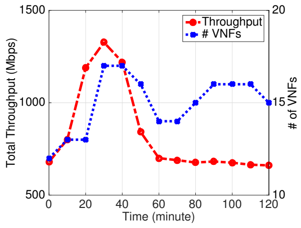

Fig. 10. Total multicast throughput and total # of VNFs over time;

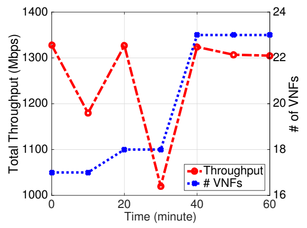

Fig. 11. Total multicast throughput and # of VNFs in case of bandwidth variation.

to 0%. Similarly, NC1 outperforms NC2 when packet loss rate is not high. So, it is desirable to produce a small number of extra coded packets for each generation in cases of high packet loss rate, and no extra coded packets if the links are reliable.

C. Evaluation of Dynamic Scenarios

We next evaluate our dynamic deployment and scaling algorithm. We generate six multicast sessions, each with a uniformly random number of receivers in the range of $[1, 4]$. The sources and receivers are distributed uniformly randomly across the six data centers in North America. The interval for collecting bandwidth, throughput and delay data is 10 minutes. The threshold values, τ, τ_1 and τ_2 are all 10 minutes, $\rho_1 = \rho_2 = 5\%$. By default, the conversion factor α is 20 (Mbps per VNF), and the max tolerable delay L_m^{max} is $150ms$, $\forall m \in M$.

1) *Throughput and # of VNFs:* We initially launch three multicast sessions. Then one extra session arrives every 10 minutes until the system contains six sessions in total. After that, one session leaves the system every 10 minutes until only three sessions remain. One receiver is added into one existing session at 70, 80, 90 minute, then one receiver leaves at 100, 110, 120 minute. We measure the throughput at each receiver to obtain the multicast throughput for each session and the total number of launched VNFs needed for fulfilling the network coding functions. In Fig. 10, the total multicast throughput, *i.e.*, $\sum_m \lambda_m$ in (2), increases for the first half an hour as the number of sessions grows and then decreases as some sessions leave the system, and the total number of VNFs also grows for the first 30 minutes, same as the total throughput, but it keeps stable for the next 10 minutes which is due to the scaling algorithm, and then decreases for another 30 minutes. In case of receiver arrivals/departures, we observe similar trend on the number of VNFs. We also notice that the total multicast throughput keeps stable from 70 to 120 minute as the arriving/leaving receivers may not affect the multicast throughput within a session as long as they are not the ones experiencing the minimum rate. This verifies that the system can launch sufficient VNFs to cater for a new demand, and also recycle the resource after it departs.

2) *Bandwidth Variation:* We investigate the performance when the system facing varying bandwidth and delays. We launch six sessions in the system at the beginning. After 10 minutes, we randomly select a current used data center, and cut inbound/outbound bandwidth of all our own VNFs in that data center by half using `netem`. We then repeat the

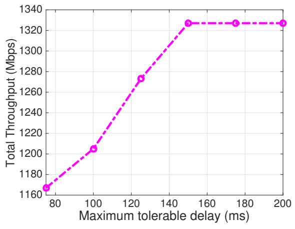

Fig. 12. Total multicast Throughput when L_m^{max} increases.

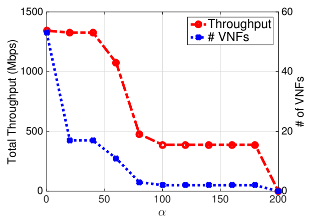

Fig. 13. Multicast throughput and # of VNFs when α increases.

bandwidth cut on a randomly selected data center every 20 minutes. We measure the total multicast throughput and the total number of VNFs needed. Results are shown in Fig. 11. We see that the throughput drops when the bandwidth is cut by half, but it recovers back in 10 minutes as the scaling algorithm notices that the low bandwidth last for $\tau_1 = 10$ minutes, and launches new VNFs to mitigate the changes. But interestingly the throughput is not recovered for the third bandwidth cut. The reason is that the scaling out operation leads to a lower objective value, so the system chooses to not scale out.

3) End-to-End Delays: Next we examine the impact of L_m^{max}. We vary L_m^{max} from $75ms$ to $200ms$ while retaining six sessions in the system and disabling the scaling algorithm. The experiment is completed within a relatively short time, during which the delay and bandwidth between data centers largely remain the same. Fig. 12 shows the aggregate multicast throughput in all sessions. The results suggest that larger L_m^{max} leads to larger throughput since the feasible paths set is enlarged. The throughput does not grow further when $L_m^{max} > 150ms$, as the newly added feasible paths do not contribute to the solution.

4) The impact of α: We now study how α influences the throughput and the number of VNFs. When $\alpha = 0$, (2) is simplified to a pure throughput maximization problem. In Fig. 13, we observe a general trend: the throughput decreases as α increases; meanwhile the number of VNFs launched for network coding functions decreases. An interesting observation is that the system refuses to launch any new VNF when $\alpha = 200$ as shown in Fig. 13. The reason is that the deployment cost increase outweighs the throughput increase. We clearly see that α should be set to a high value if the system is cost-sensitive, while a smaller α is preferred if it is performance (throughput) sensitive.

5) Delay Overhead for VNF Launch and Update: We then investigate the delay overhead to launch or update a network coding function. We evaluate the time needed in three cases, with VNFs running in the EC2 data center in Oregon: i) launch a new VM instance; ii) start a network coding function on a launched VM; iii) update the forwarding table in a running VNF. A forwarding table with ten entries is used to ii) and iii) above. We measure the delays for ten times. Average results are $35s$ for i) and $376.21ms$ for ii), *i.e.*, launching a new instance is the most time consuming (100× slower than starting a new network coding function on a launched VM). This justifies our design that after receiving a `NC_VNF_END` message, a VNF remains alive for τ time before shutting down. The relaunching cost is saved if the VNF is needed again within τ time. Tab. III shows that the average time increases from $78.44ms$ to $310.61ms$ when the forwarding table update percentage rises from 20% to 100%. The delays incurred in case ii) and iii) are relatively small as the VNF adjustment occurs at intervals of 10 minutes. Overall, we conclude that the performance is satisfactory when using our network coding VNFs and the scaling algorithm in practical scenarios.

TABLE III
TIME OVERHEAD FOR FORWARDING TABLE UPDATE

Update Percentage	20%	40%	60%	80%	100%
Average Time (ms)	78.44	145.82	194.06	264.82	310.61

VI. CONCLUSION

Network coding is a powerful technique that can achieve high throughput and lower routing complexity. We propose implementing network coding as a virtual network function in geo-distributed clouds on the Internet, following the NFV paradigm. We carefully build the virtual network coding functions upon the latest techniques in NFV, and design efficient algorithms for dynamic coding function deployment and scaling. We deploy our system across geo-distributed clouds in Amazon EC2 and Linode, and carry out extensive experiments to evaluate the efficacy, robustness and overhead. Results show that the proposed system achieves substantially higher throughput than a non-network coding based system, while only incurring small additional delay overhead. We believe that our attempt is an important step towards unleashing the full potential of network coding in practical Internet applications. We believe our work may shed light on the design and implementation of other dynamic Internet applications/services using the NFV paradigm, in terms of geo-distributed dynamic network function deployment and scaling. Modularizing the system design is a possible future direction to explore, so that our system can directly support a broad range of application scenarios beyond network coding, once the network coding related modules are replaced by other application-specific modules.

REFERENCES

[1] R. Ahlswede, N. Cai, S. Y. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, Jul 2000.
[2] P. A. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Allerton Conference on Communication, Control, and Computing*, October 2003.
[3] P. Chou and Y. Wu, "Network Coding for the Internet and Wireless Networks," *IEEE Signal Processing Magazine*, vol. 24, no. 5, pp. 77–85, Sept 2007.
[4] Z. Li, B. Li, and L. C. Lau, "On Achieving Maximum Multicast Throughput in Undirected Networks," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2467–2485, June 2006.
[5] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive View of a Live Network Coding P2P System," in *Proc. of ACM IMC*, 2006.
[6] C. Gkantsidis and P. R. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. of IEEE INFOCOM*, 2005.
[7] C. Wu, B. Li, and Z. Li, "Dynamic Bandwidth Auctions in Multioverlay P2P Streaming with Network Coding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 6, pp. 806–820, June 2008.

[8] R. Yu, G. Xue, V. Kilari, and X. Zhang, “Network Function Virtualization in the Multi-tenant Cloud,” *IEEE Network*, vol. 29, no. 3, pp. 42–47, May 2015.

[9] *Amazon EC2 Service Level Agreement*, https://aws.amazon.com/ec2/sla/.

[10] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, “Scalable Application Layer Multicast,” in *Proc. of ACM SIGCOMM*, 2002.

[11] *Intel Data Plane Development Kit: Programmers Guide*. Intel Corporation, Mountain View, CA, USA, 2016.

[12] S. E. Deering and D. R. Cheriton, “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110, May 1990.

[13] S. Deering, “RFC1112 Host Extensions for IP Multicasting,” *https://tools.ietf.org/html/rfc1112*, 1989.

[14] Y.-h. Chu, S. G. Rao, and H. Zhang, “A Case for End System Multicast,” in *Proc. of ACM SIGMETRICS*, 2000.

[15] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, “ALMI: An Application Level Multicast Infrastructure,” in *Proc. of USENIX USITS*, 2001.

[16] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole, Jr., “Overcast: Reliable Multicasting with on Overlay Network,” in *Proc. of USENIX OSDI*, 2000.

[17] P. Li, S. Guo, S. Yu, and A. V. Vasilakos, “Reliable Multicast with Pipelined Network Coding Using Opportunistic Feeding and Routing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3264–3273, Dec 2014.

[18] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li, “Celerity: A Low-delay Multi-party Conferencing Solution,” in *Proc. of ACM Multimedia*, 2011.

[19] Y. Feng, B. Li, and B. Li, “Airlift: Video Conferencing as a Cloud Service using Inter-Datacenter Networks,” in *Proc. of IEEE ICNP*, 2012.

[20] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, “ClickOS and the Art of Network Function Virtualization,” in *Proc of USENIX NSDI*, 2014.

[21] J. Hwang, K. Ramakrishnan, and T. Wood, “NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms,” *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[22] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, “Making Middleboxes Someone else’s Problem: Network Processing As a Cloud Service,” in *Proc. of ACM SIGCOMM*, 2012.

[23] J. Elias, F. Martignon, S. Paris, and J. Wang, “Optimization Models for Congestion Mitigation in Virtual Networks,” in *Proc. of IEEE ICNP*, 2014.

[24] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, “Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud,” *CoRR*, vol. abs/1305.0209, 2013. [Online]. Available: http://arxiv.org/abs/1305.0209

[25] B. Khasnabish, S. Sivakumar, E. Haleplidis, and C. Adjih, “Impact of Virtualization and SDN on Emerging Network Coding,” *https://tools.ietf.org/pdf/draft-khasnabish-nwcrg-impact-of-vir-and-sdn-04.pdf*, 2015.

[26] D. Szabó, F. Németh, B. Sonkoly, A. Gulyás, and F. H. Fitzek, “Towards the 5G Revolution: A Software Defined Network Architecture Exploiting Network Coding As a Service,” in *Proc. of ACM SIGCOMM*, 2015.

[27] D. Szabó, A. Csoma, P. Megyesi, A. Gulyás, and F. H. P. Fitzek, “Network Coding as a Service,” *CoRR*, vol. abs/1601.03201, 2016. [Online]. Available: http://arxiv.org/abs/1601.03201

[28] B. Chandrasekaran and T. Benson, “Tolerating SDN Application Failures with LegoSDN,” in *Proc. of ACM HotNet*, 2014.

[29] N. Katta, H. Zhang, M. Freedman, and J. Rexford, “Ravana: Controller Fault-Tolerance in Software-Defined Networking,” in *Proc. of ACM SOSR*, 2015.

[30] T. Ho, M. Medard, J. Shi, M. Effros, and D. R. Karger, “On Randomized Network Coding,” in *Proc. of 41st Annual Allerton Conference on Communication, Control, and Computing*, 2003.

[31] M. V. Pedersen, J. Heide, and F. H. P. Fitzek, “Kodo: An Open and Research Oriented Network Coding Library,” in *Proc. of IFIP Networking*, 2011.

[32] “netfilter,” www.netfilter.org/.

[33] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, “Guaranteeing Deadlines for Inter-datacenter Transfers,” in *Proc. of ACM EuroSys*, 2015.

[34] “Enabling Enhanced Networking on Linux Instances in a VPC,” *Amazon AWS Documentation*, 2016. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[35] Y. Dong, Z. Yu, and G. Rose, “SR-IOV Networking in Xen: Architecture, Design and Implementation,” in *Proc. of USENIX WIOV*, 2008.

[36] L. R. Ford and D. R. Fulkerson, “Maximal Flow through a Network,” *Canadian Journal of Mathematics*, vol. 8, no. 3, pp. 399–404, 1956.