IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier N/A

Practical Evaluation of Multi-source Coded Downloads

PATRIK J. BRAUN[1,2], MURIEL MÉDARD[1] AND PÉTER EKLER[2]

[1]Research Laboratory of Electronics (RLE), Massachusetts Institute of Technology (MIT), Cambridge, MA 02139 USA (email: {pbraun, medard}@mit.edu)
[2]Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, 1111 Hungary (email: {patrik.braun, peter.ekler}@aut.bme.hu)
This work has been done, when Patrik J. Braun was a visiting student researcher at MIT.

Corresponding author: Patrik J. Braun (e-mail: pbraun@mit.edu).

This work was performed in the frame of FIEK_16-1-2016-0007 project, implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the FIEK_16 funding scheme. It was also supported by the BME-Artificial Intelligence FIKP grant of EMMI (BME FIKP-MI/SC), by the János Bolyai Research Fellowship of the Hungarian Academy of Sciences and by the Fulbright and Rosztoczy Foundation Scholarship programs.

ABSTRACT In this paper, we introduce two multi-source download protocols for loosely orchestrated networks that have high potential in Information-Centric Networking (ICN). We focus on services with high bandwidth and low delay requirements, such as video streaming. We propose MUlti-source Transmission Protocol (MUTP) for uncoded multi-source data delivery and extend it with network coding capabilities to create Coded MUTP. We investigate their throughput using a custom-designed system that includes browser extensions and proxy servers. The browser extensions intercept YouTube video downloads and forward them through our proxy server, using parallel HTTP requests, Uncoded MUTP or Coded MUTP approach. We present measurement results collected in 2018-2019, over eleven months that include 1,300,000 log records from more than 960 GBs of video download. We show that even when downloading from only two sources, our protocols can match the heavily optimized HTTP. Furthermore, by increasing the number of sources to four or higher, MUTP protocols can outperform HTTP, reaching an up to three-fold goodput (useful throughput) increase. In addition, we show that the proposed solution avoids the straggler problem, therefore adding more sources to a network increases its goodput.

INDEX TERMS Edge cloud, JavaScript, multi-source download, Network coding, video download, WebRTC

I. INTRODUCTION

In a multi-source download, a single receiver downloads the same data from multiple sources. This has high potential in Information-Centric Networking (ICN) [1], especially in video streaming applications [2].

Video streaming accounted for 60% of the mobile Internet traffic in 2018 [3]. Compared to fixed-line broadband networks, the main advantage and also the challenge of mobile Internet connections is their mobility. On the one hand, users can watch a YouTube, Netflix, or live steam video while commuting to work. On the other hand, the majority of services that are used over the mobile network are mostly based on a traditional client-server setup, through protocols like HyperText Transfer Protocol (HTTP) [4]. In this networking scenario, the client receives data from exactly one server. Furthermore, the servers are usually placed in the core of the network, far away from the client. When a user travels on a train or in a car with high speed, their mobile connection can have bandwidth fluctuations, because of signal losses

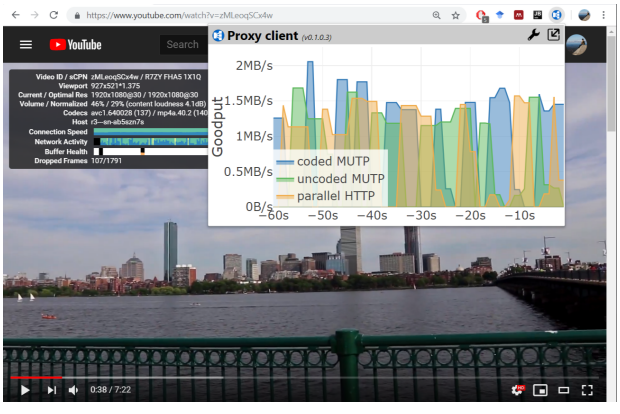

FIGURE 1. Screenshot of downloading a YouTube video with our multi-source MUTP protocols and with parallel HTTP requests.

or handovers [5]. These fluctuations can lead to a reduction in video quality or even to stream interruptions [6]. The interruptions may get worse when seeking or changing the played content since the application cannot use its pre-cached buffer to overcome temporary communication errors.

A possible solution to this issue is to use ICN with in-network caching or to create an edge cloud system [7] [8]. In both solutions, the content gets cached to the nearby network infrastructure that can serve as source nodes. If clients can connect to multiple cell towers, they can download from all of them simultaneously. This minimizes the effect of having a single weak connection. To achieve multi-source download, conventional protocols are not sufficient. Furthermore, in a mobile video streaming scenario, the available nearby source nodes are changing as the user travels. Therefore, it is challenging or not feasible to coordinate all source nodes such that they do not send the same packets to the receiver.

There have been several works on multi-source download: Multi-source content delivery through multipath transmission in ICNs was modeled by Hashmeni and Bohlooli [1]. They selected virtual round-trip time (VRTT) as a key parameter of performance evaluation. They estimated VRTT in their work and use it to calculate the network throughput. Miyoshi et al. proposed a congestion control mechanism for Content-Centric Networking (CCN) with multi-source content retrieval [9]. They used end-to-end flow control to regulate the transmission only on the congested paths. Multi-source and multipath File Transfer Protocol (mmFTP) for ICN networks was proposed by Thomas et al. [10]. Their measurement-based results showed that mmFTP might have a 37% throughput increase compared to a single-source download, while it avoids congested paths or sources. Bruneau et al. proposed MS-streaming, a multiple-source streaming solution that splits video into multiple independent sub-streams and offers methods for bit rate adaptation and server switching [11]. Compared to optimal Dynamic Adaptive Streaming over HTTP (DASH) systems, MS-streaming can achieve up to a 74% mean bit rate gain. Batalla et al. investigated station-to-device and device-to-device media streaming methods in a smart city environment for future 5G networks [12]. They proposed a DASH extension called Multiple Description - Dynamic Adaptive Streaming (MD-DASH) with full backward compatibility. Their solution encodes the same movie with H.264 and H.265 codecs into different bit rates that are downloaded simultaneously from multiple sources. The authors showed that their solution could exploit the benefits of multiple sources and achieve a significant performance improvement compared to unipath approaches. Pucha et al. propoesd Similarity-Enhanced Transfer (SET), a file handprinting solution to tag the similarities in different files [13]. SET is aimed to improve data availability, and thus the network throughput in distributed systems, by downloading data from multiple sources. Once the files are tagged, SET can reach up to a 30% bandwidth gain compared to an equivalently configured BitTorrent system.

We have previously shown that Random Linear Network Coding (RLNC) [14] may also be used to improve the throughput of a multi-source network [15]. RLNC creates linear combinations of the original packets using random coefficients. These coefficients are chosen from a sufficiently large finite field so that the coded packets are linearly independent with a high probability. The main advantage of RLNC is that it is a rateless code. Thus, in case of packet loss, new packets can be generated without changing the coding configuration. Sundararajan et al. introduced a network coded approach to Transmission Control Protocol (TCP) and showed that their scheme achieves a much higher throughput compared to TCP over a lossy link [16]. They proposed a sliding window network coding approach, where they used feedback to adjust the window of packets that they coded on. Kim et al. introduced a model to analyze the performance of TCP with network coding [17]. They showed that network coding could prevent TCP's performance degradation that often can be observed in lossy networks. Sørensen et al. have presented Network Coded Filesystem Shim (NCFSS), a filesystem-level solution for multipath, and multi-source download with RLNC [18]. They provided a proof-of-concept implementation of their proposed solution and showed that it improves access and download time by a factor of two to five compared to downloading from a single source. In our work, we design and implement an RLNC-based protocol for multi-source download and compare its goodput with an uncoded multi-source protocol and with a Parallel HTTP-based approach.

The straggler problem is also a challenge [19] [20] in distributed systems. The network throughput may drop if the client has to wait for a packet that is unusually late to arrive. In this paper, we propose a solution that avoids the straggler problem.

a: Main contributions

In this paper, we extend our previous works on coded multi-source download [15] [21]. We propose two protocols for multi-source download and evaluate their goodput through measurements. The main contribution of this paper can be summarized as follows:

- Section II describes the problem formally.
- Section III presents our proposed protocols for multi-source download. The first protocol is the MUlti-source Transmission Protocol (MUTP) that transfers uncoded packets from several servers to one client. We also describe a testbed in this section that can intercept YouTube video downloads. The testbed downloads the intercepted video through several servers, using one of our protocols or a simple parallel HTTP requests-based approach that starts multiple HTTP downloads for the same data and chooses the fastest among them. FIGURE 1 shows an example of our solution downloading a YouTube video.
- Section IV describes the measurement preparation and setup. We ran our testbed in the Amazon Cloud with

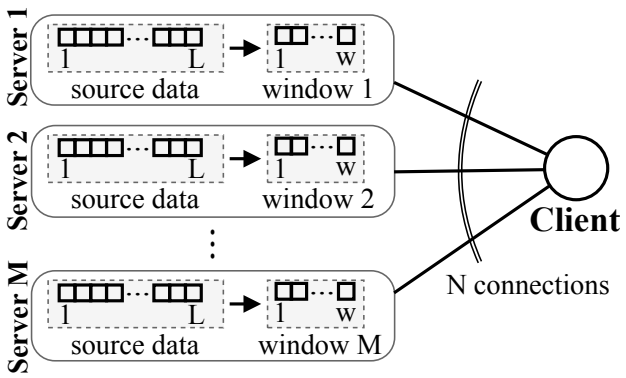

FIGURE 2. Multi-source download scenario with M servers and $N \leq M$ connections.

18 servers located in Europe and the USA over eleven months.

- Section V presents our results.
- Section VI summarizes the results and describes our future research plans.

b: Significance of our work

Our solution differs from previous work in three main aspects. 1) We focus on loosely-orchestrated networks where the servers cannot cooperate. 2) We propose protocols that can be applied in the transportation layer or over User Datagram Protocol (UDP) in the application layer. The protocols also avoid the straggler problem. 3) We use RLNC as part of the protocol.

The significance of our work is that we show that our uncoded MUTP protocol outperforms the Parallel HTTP solution. Applying rateless RLNC encoding on the transmitted data further increases goodput. We obtained these results by analyzing more then 1,300,000 log records, where each record represents one video chunk download. The log records were obtained by running an extensive measurement campaign for eleven months in Europe and the USA.

II. PROBLEM DEFINITION

In this paper, we focus on a scenario that has M servers and one client as FIGURE 2 shows. All M servers contain the same L original data packets that the client would like to download. The client connects to $N < M$ nodes and starts to download the original data. Connections between the client and the servers are unreliable in both directions.

We measure the client progress with Degrees of Freedom (DoF). DoF increases by one if the client receives a new, useful packet. The client sends cumulative feedback that contains information about all of its previously received packets. It is not required to acknowledge each packet separately. Therefore, the client may also decide to skip sending feedback for some of the received packets. There is no constraint on the frequency of sending feedback during download, but once the client has all L original data packets, it should send a feedback packet to indicate that the download is ready. If a feedback packet gets lost, we consider the event to be the same as if the client skipped sending it.

In this paper, we focus on a loosely orchestrated scenario. The server nodes do not have information about each other, i.e., they do not know how many nodes the client is connected to, and the bandwidth of the nodes is also not available for packet scheduling. For packet scheduling, a server must rely on two information: 1) the previously sent packets, 2) the information from the feedback. Note that the feedback from the client to the server is delayed. Thus, the servers never have full information about the network.

Servers maintain a window of size $w \leq L$ to limit the memory needed for transmission.

In a conventional sliding window approach, packets with the lowest packet *id* are chosen from the window for transmission. If a packet with the lowest *id* gets successfully transmitted, it can be removed from the window, and the window can slide to include new packets. In a multi-source scenario, we cannot use this conventional sliding window, since in that case, all servers would send the same packet. Therefore, in this paper, we consider a *strict moving* window setup. A server may schedule any packets from its window. A packet can be removed from the window if the client successfully received and acknowledged its reception. To have a constraint on the packet delay, we define $\mathcal{W}(t)$, the set of packets in the window at time t the following way:

$$\begin{aligned} \mathcal{W}(t) &= \{i \in \mathcal{L} \mid w_{\min} \leq i < w_{\min} + w\} \\ \mathcal{L} &= \{0, \ldots, L\} \\ \mathcal{L}_{\text{acked}}(t) &= \{i \in \mathcal{L} \mid \text{ packet } i \text{ was acknowledged by time } t\} \\ w_{\min} &= \min(\mathcal{L} \backslash \mathcal{L}_{\text{acked}}(t)), \end{aligned} \tag{1}$$

where $\mathcal{L}$ is the set of original data packets, $\mathcal{L}_{\text{acked}}(t)$ is the set of all successfully received and acknowledged packets by time t and $w_{\min}$ is the not-yet-acknowledged packet with the lowest index. An example of the strict moving window is shown in FIGURE 3.

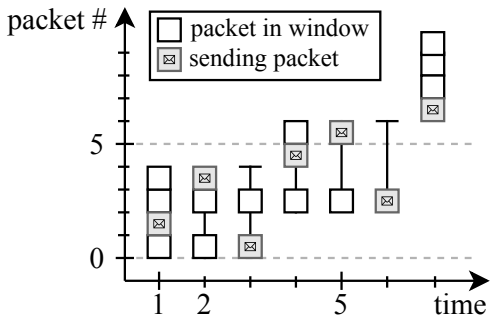

FIGURE 3. Strict moving window example for window $w = 4$, assuming 0 round trip time and that the connection between the server and the client is reliable.

In this paper, we focus on finding the achievable maximum goodput (useful throughput) of a system that fulfills the model that is presented in this section.

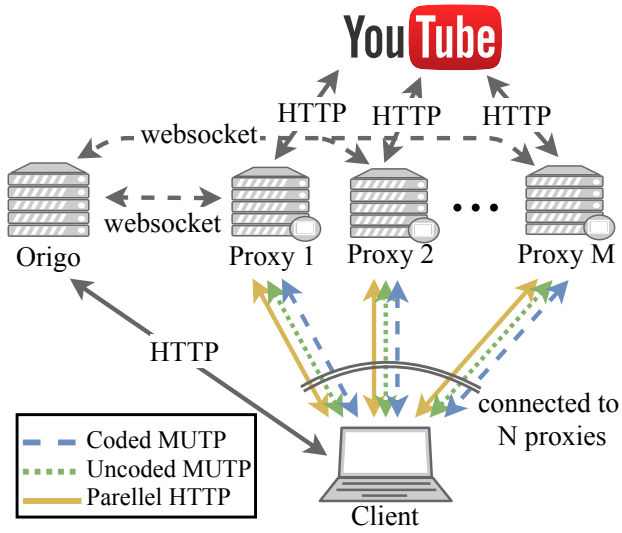

FIGURE 4. System overview.

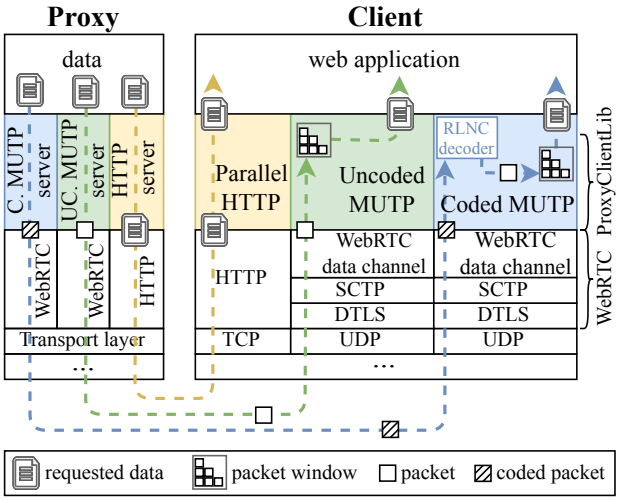

FIGURE 5. Network stack.

III. SYSTEM DESCRIPTION

We propose two protocols for multi-source download in loosely orchestrated network scenarios, where data source cannot cooperate. We design the MUlti-source Transmission Protocol (MUTP) for uncoded data transfer from multiple sources. Based on MUTP, we propose Coded MUTP with network coding for encoded multi-source data transfer. We also refer to these protocols as Uncoded and Coded MUTP to emphasize their differences.

We also design a system that supports Uncoded and Coded MUTP downloads. FIGURE 4 shows our system setup, consisting of an *Origo* server and several proxy servers and clients. The responsibility of the *Origo* server is to manage the proxy servers, serve as the entry point to the system, and receive metadata messages from the client and proxy servers, such us statistical log data or error reports. The proxy servers use WebSocket [22] with socket.io[1] to connect to the *Origo* server. Through this connection, the proxies periodically send status updates to the *Origo* server. Each client uses HTTP requests to download the list of the available proxies from the *Origo* server and to send metadata to the *Origo* server.

We have designed a JavaScript library, ProxyClientLib, which runs on the client. ProxyClientLib connects to N proxies and requests the same data from all of them. To achieve this, the client sends the URL of the requested data to the proxies. Each proxy server downloads data from the URL. ProxyClientLib can download the content from the proxies in three different ways: over a simple*Parallel HTTP*, or using Uncoded MUTP or Coded MUTP protocols. FIGURE 5 shows the network stack for the three different approaches. *Parallel HTTP* sends the same HTTP request over TCP to all connected proxies and uses the fastest response as the result of the download (once the data is obtained the other HTTP connections are terminated). In the case of *Parallel HTTP*, feedback and re-transmissions are handled by the underlying protocol. Uncoded MUTP and Coded MUTP use Stream Control Transmission Protocol (SCTP) protocol in unreliable mode over the data channel of Web Real-Time Communication (WebRTC) to connect to the proxies. We chose WebRTC because it is the only way to create an unreliable connection from JavaScript without the need of installing any third-party application to the user's machine.

We have also designed browser extensions for Firefox and Chrome that intercept YouTube video downloads and use ProxyClientLib to download video over all three ways. We have used JavaScript and browser extensions to make it as convenient as possible for our users to participate in this research. FIGURE 1 shows a screenshot of our Chrome extension.

A. MULTI-SOURCE TRANSMISSION PROTOCOL (MUTP)

With MUTP protocol, proxies slice the original data into L packets of 1100 bytes[2]. Each proxy maintains two lists: *in-window* packets, and *in-transit* packets. In-transit packets are those that have been sent, but no feedback has yet been received.

The client maintains a list of *received* packets that increased its DoF (DoF increases at the client if it receives a packet with an ID that was not present in its *received* list). The client sends cumulative feedback based on its *received* list. It may also decide to skip sending some of the feedback.[3] The frequency that the client sends feedback is a parameter of our testbed.

Based on the obtained *received* list from the client and the *in-transit* list, each server creates a *sendable* list of packets.

[1] socket.io: https://socket.io/

[2] According to our observation, packets bigger than 1100 bytes over WebRTC get fragmented in the IP layer that results in throughput drop.

[3] Sending a feedback packet in the application layer with WebRTC triggers an acknowledgment to that feedback in the lower network layers. This behavior significantly reduces throughput, and the system performs better if some of the feedback packets are skipped. We set the feedback frequency empirically.

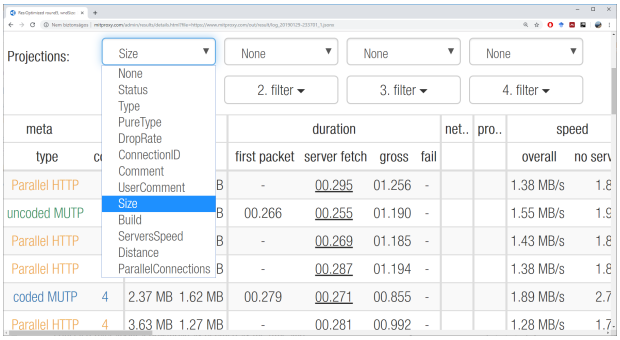

FIGURE 6. Screenshot about the andministrator page of our testbed, showing a list of measurement results.

The servers use this to choose a packet for transmission. Since the proxies cannot communicate with each other, optimal scheduling is not possible. Therefore we implement a random scheduler, that chooses a packet uniformly at random from the *sendable* list without replacement.

B. CODED MUTP

Coded MUTP uses a similar approach as Uncoded MUTP to transmit packets, but instead of sending the original packets, it uses random linear network coding (RLNC) to create coded packets. To achieve this, Coded MUTP first organizes the original L packets into g sized groups, called *generations*. Following this, the packets within a *generation* are linearly combined over a given finite field. Compared to the uncoded approach, servers keep *generations* in their window instead of individual packets. We use Kodo [23] for RLNC encoding over the field size of 2^8. Kodo is a C++ library that we compile to JavaScript with emscripten[4].

Similarly to the Uncoded MUTP, the Coded MUTP client also tracks the received DoF in a *received* list, but it does this at the generation level: the DoF of a generation is the number of received, linearly independent packets of that generation. The client sends this *received* list to the server as cumulative feedback. To keep the comparison fair, Coded MUTP sends feedback with the same frequency as Uncoded MUTP.

Servers also keep track of the packets in transit per *generation*. Servers schedule packets based on their *in-transit* and *received* list (from the client's feedback). We use a *rarest generation first* approach for packet scheduling, based on the *rarest piece first* algorithm of BitTorrent [24]. *Rarest generation first* sends a packet from a *generation* that has the least received DoF and in transiting packets. We chose this method to schedule a *generation* for sending because it has already shown potential to improve throughput in RLNC enhanced distributed systems [25] [15].

C. SYSTEM CONFIGURATION AND DATA PROCESSING

Our system has more than 60 different configuration options, including the number of connected proxies, window size, and generation size. To measure the impact of changing the configuration, we collect 58 basic characteristics of a single download, including gross downloaded data and duration. Based on these, we further derive 14 characteristics, like throughput or goodput. We have also developed a detailed administration website to follow the status of our proxies. The website also provides a robust toolbox for analyzing data live, right after it is collected, without the need of any post-processing. Among the 58 basic characteristics, we have collected data about a download as detailed as the time and the originating proxy of each downloaded packet (from the perspective of the client). FIGURE 6 shows a screenshot about the administration website.

We used this tool to fine-tune our measurements and to obtain a quick insight into our system.

[4]emscripten: https://kripken.github.io/emscripten-site/

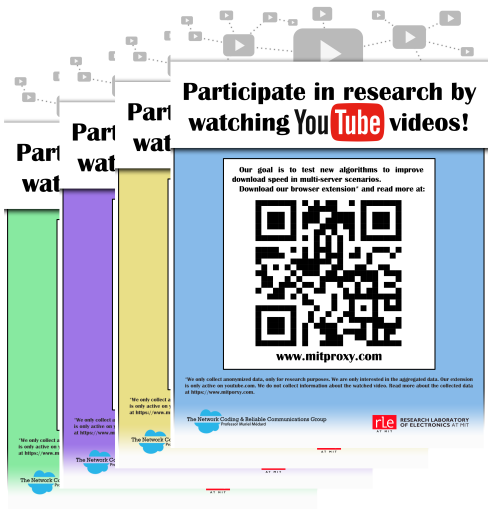

FIGURE 7. Posters for our measurement campaign.

IV. MEASUREMENT PREPARATION

Before starting our measurements, we ran our system for more than two months in beta mode with limited users to find the best configuration (like window size, generation size, feedback sending frequency) for our measurement and to discover improvement possibilities in our implementation.

For our main measurements campaign, we used Docker[5] containers in Amazon Web Services (AWS)[6] to host one *Origo* server and 18 proxy servers. The servers were distributed among five locations: Virginia, Ohio, and Oregon in the USA and Frankfurt and Paris in Europe. We have uploaded our browser extension to Chrome Web Store[7] and to Firefox add-ons[8] with the name *RLNC Proxy client*. We have also created an official website (www.mitproxy.com) for the

[5]Docker: https://www.docker.com/
[6]Amazon Web Services: https://aws.amazon.com/
[7]Chrome extension: https://chrome.google.com/webstore/detail/rlnc-proxy-client/jgkegjhffajllgamghkdopkeabjbclfh
[8]Firefox extension: https://addons.mozilla.org/firefox/addon/rlnc-proxy-client

project and asked visitors to use our extension. Apart from setting up the website, we run an advertisement campaign to get publicity for our research. FIGURE 7 shows posters that were distributed at MIT, USA. Furthermore, we also contacted European universities like BME (Hungary), and TU-Dresden (Germany) and asked their students to participate.

Our measurements run between in June 2018 and April 2019. Throughout these months, there were more than 75 extension installs and more than 25 active weekly users. Our system collected more than 1,300,000 log records (each record represents one download) that were generated by watching more than 960 GBs of YouTube videos. Since our system uses HTTP over TCP and SCTP over WebRTC based connections and the achievable throughput of a WebRTC connection is significantly lower than an HTTP over TCP connection [26], we decided to limit the bandwidth of proxy connections to make the performance of the protocols comparable. Furthermore, this is a better representation of the multi-source scenario, when multiple connections are needed to utilize the available download bandwidth at the client fully. We limited 14 of our proxies to 896 KB/s and four proxies to 1,792 KB/s. We used the built-in linux commands *qdisc*[9] and *iptables*[10] for in- and outbound traffic shaping.

As we described in Section I and II, we focus on loosely orchestrated scenarios where the number of connected proxies and their bandwidth are not known in advance, so this information cannot be used for packet scheduling. We emulate this behavior by having the clients connect to proxy servers at random. A client randomly chooses $N \in \{1, 2, 4, 6\}$ among the available 18 proxies to connect to. Furthermore, we restricted our extension to only connect to proxies within 3,000 km of the user to avoid connections with a high round trip time (RTT).

V. RESULTS

Throughout our measurement campaign, we obtained our results in three steps: 1) first, we investigated our protocols in details, by observing the arrival of the individual packets at the client. 2) then we measured the distribution of YouTube video chunk size[11]. 3) Finally, we evaluated the collected data, focusing on the measurements results about the most common chunks sizes.

A. UNCODED AND CODED MUTP PROTOCOL INSIGHT

FIGURE 8 and FIGURE 9 show an example for packet arrival and feedback (ACK) timing for downloading 1.92 MB with Uncoded MUTP and Coded MUTP, respectively. The figures show that during this sample run, Coded MUTP received significantly less duplicate packets (i.e., packets that do not increase DoF at the client) and could finish the download significantly earlier. On the other hand, there is a gap between the download finish and the arrival of the late (*after-finishing*) packets. This gap corresponds to the post-processing of the downloaded packets. During this post-processing, our application cannot handle any incoming packets, as JavaScript is single-threaded. Therefore the after-finishing packets are handled right after processing is finished. As FIGURE 9 shows, the post-processing gap is larger for the Coded MUTP. Coded MUTP needs to decode the RLNC encoded packets and also concatenate them, while the Uncoded MUTP only needs to concatenate them. Furthermore, the figures give a good insight into the applied *strict moving* window mechanism and the distribution of duplicate packets over time. In the case of Uncoded MUTP, the duplicates arrive throughout the whole download. With Coded MUTP, the effect of the rarest first generation approach can be observed, as the duplicate packets arrive in a burst at the end of each RLNC generation.

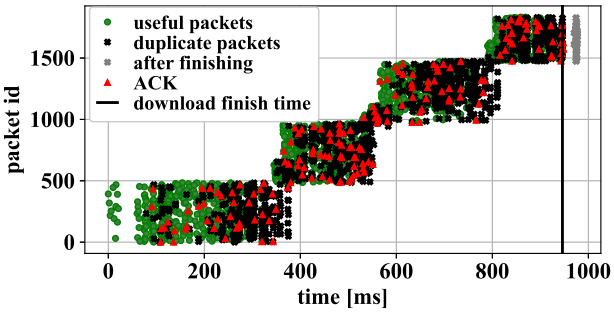

FIGURE 8. Sample of packet arrivals while downloading a 1.92 MB chunk with Uncoded MUTP (window size $w = 360$).

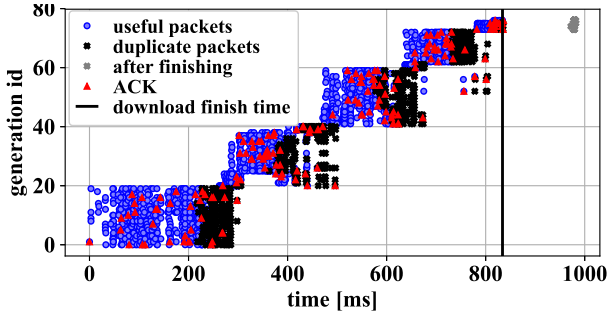

FIGURE 9. Sample of packet arrivals while downloading a 1.92 MB chunk with Coded MUTP (window size $w = 360$, generation size $g = 24$).

B. YOUTUBE VIDEO CHUNK SIZE DISTRIBUTION

FIGURE 10 shows statistics about our collected data based on the downloaded YouTube video chunk size. We have observed data chunks between a few KBs to 5 MB, but most of our data lie in the range of 1 MB - 2 MB. Therefore, our evaluation focuses on this range.

C. SYSTEM GOODPUT

In our results, we compare the *throughput*, *goodput* and *normalized goodput* of different setups. We define these

[9] tc qdisc manual page: https://linux.die.net/man/8/tc

[10] iptables manual page: https://linux.die.net/man/8/iptables

[11] According to our observation, a YouTube video is downloaded through several smaller data chunks, which size varies between approximately 1 KB to 5 MB.

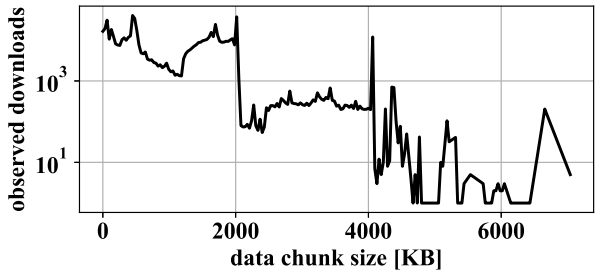

FIGURE 10. YouTube video chunk size distribution.

quantities the following way:

$$\begin{aligned} \text{throughput [byte/s]} &= \frac{\text{gross download [byte]}}{\text{duration [s]}} \\ \text{goodput [byte/s]} &= \frac{\text{net download [byte]}}{\text{duration [s]}} \\ \text{normalized goodput} &= \frac{\text{goodput [byte/s]}}{\text{throughput [byte/s]}} \end{aligned} \tag{2}$$

We use the *normalized goodput* to compare the amount of received packets that increase the DoF at the client to all received packets. We introduce *normalized goodput* to have a better comparison between Parallel HTTP and the MUTP protocols since the throughput of a WebRTC based protocol is significantly lower than the throughput of HTTP as FIGURE 11 also shows for downloading data with one connection.

We present our result in a grouped boxplot arrangement. For each setup (each tick on the x-axis), we present a boxplot for all three download approach. Each boxplot contains data at least from 100 video chunk downloads. The box part of the boxplot is the interquartile range (IQR) that represents data between Q1: 25 percentile and Q3: 75 percentile, while the horizontal line on the box is the median (i.e., Q3: 50 percentile). The whiskers are at Q1 $-$ 1.5 $*$ IQR and at Q3 $+$ 1.5 $*$ IQR. The circles outside the whiskers are outliers.

a: Downloading 1-2MB chunks

FIGURE 11 shows combined results of downloading 1-2MB sized chunks from $N \in \{1, 2, 4, 6\}$ connection with different upload bandwidth. Results show that the throughput of all three approaches increases continuously as N grows. This characteristic comes from the fact that in our measurements, the client has usually higher download rate than the combined upload rate of the servers. In contrast, the goodput of Parallel HTTP is approximately equal to the fastest upload rate of the servers. This shows that Parallel HTTP operates with significant overhead on the network. Furthermore, we observe a slight goodput decrease in the HTTP connection with the increase of N. In our interpretation, this is caused by the client's connection getting saturated, thus resulting in a connection with a reduced bandwidth to the fastest server. These results show that in the investigated scenario, the simple Parallel HTTP can utilize the upload rate of the fastest server, but with the cost of a significant overhead on the network.

The two MUTP protocols have approximately the same performance regarding goodput if $N = 1$. Increasing N, Coded MUTP has a slightly better mean goodput than Uncoded MUTP, while Uncoded MUTP has slightly better throughput than Coded MUTP. The lower throughput for the Coded MUTP originates from the RLNC calculation overhead. This overhead decreases the packet send rate at the Coded MUTP, but using RLNC increases the chance that the received packet will be useful (i.e., increases the DoF at the client). The goodput mean results show that, even with fewer sent packets, Coded MUTP outperforms Uncoded MUTP regarding the received useful data per second.

Comparing the MUTP protocols to Parallel HTTP, we observe that using the random packet scheduling approach for Uncoded MUTP significantly increases the probability that a received packet is useful. The goodput of the MUTP protocols reaches the goodput of the parallel HTTP in case of $N = 2$, and they significantly outperform the HTTP-based approach with $N \geq 6$. Furthermore, using only two 896KB/s connections, the MUTP connections outperform Parallel HTTP regarding normalized goodput. The gain of our protocols further increases as N increases, compared to the HTTP-based approach. This shows that our protocols avoid the straggler problem since the newly added sources do not limit the network goodput.

Our current implementation of the MUTP protocols cannot optimally utilize the extra bandwidth that a faster server provides in case of in heterogeneous network, as FIGURE 11 shows this in column 1:1,792KB/s, 3:896KB/s and 4:896KB/s. This characteristic is caused mainly by WebRTC and could have been avoided if we had full control over the underlying protocol.

Normalized goodput shows that if all three download approaches have the same packet send rate, Uncoded MUTP has an up to two-fold performance increase compared to Parallel HTTP. Furthermore, Coded MUTP has an up to three-fold performance increase compared to Parallel HTTP and a 25% performance increase compared to Uncoded MUTP.

b: Normalized goodput with regards to data size

We have investigated the normalized goodput of our system with one 1,792 KB/s and three 896 KB/s connections, based on the downloaded data size as shown in FIGURE 12. Throughput results show that with larger chunk size, the congestion control in the underlying protocols has time to speed up. Results show that the MUTP protocols increase their goodput at a higher rate as the chunk size increases. As we compare the received useful packet to all received packets in the normalized goodput figure, we observe that the approaches show stable performance. We also observe a slight increase in normalized goodput as chunk size increases that come from the mentioned congestion control speed up of the underlying protocols. It is also important to note that Coded MUTP significantly outperforms Parallel HTTP,

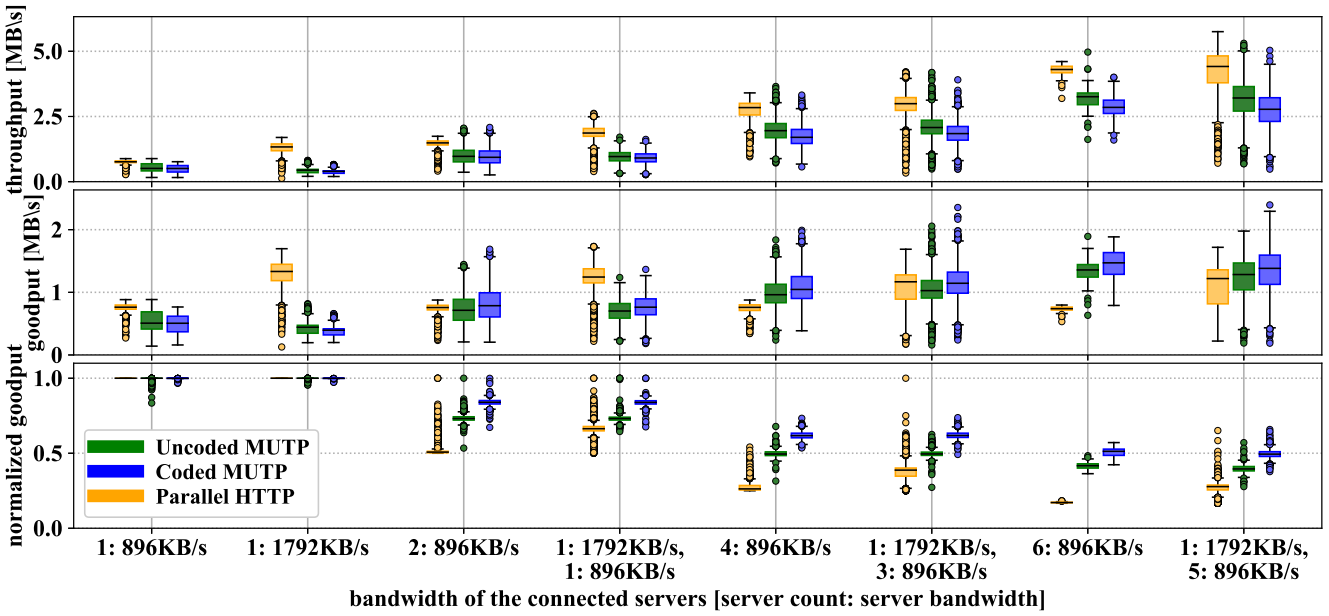

FIGURE 11. Grouped boxplot representation of downloading 1-2MB data from $N \in \{1, 2, 4, 6\}$ servers with 896 KB/s and 1,792 KB/s upload bandwidth with window size $w = 240$ and generation size $g = 24$.

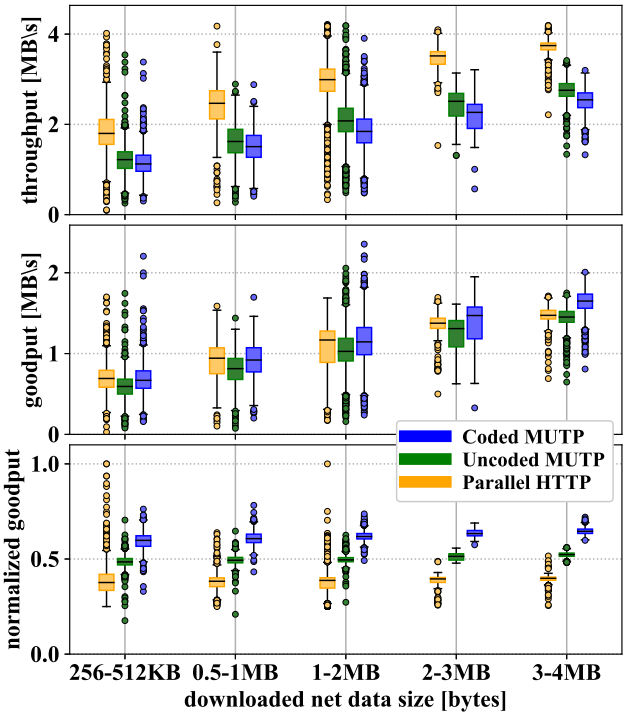

FIGURE 12. Normalized goodput for downloading video data chunks with different size over one 1,792 KB/s and three 896 KB/s connections with window size $w = 240$ and generation size $g = 24$.

starting from the 256-512 KB range. This shows that using RLNC can be beneficial even for small data transfers.

c: Normalized goodput for generation size $g \in \{12, 20, 24\}$

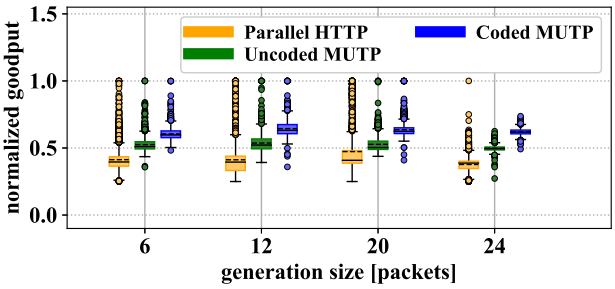

FIGURE 13. Normalized goodput of downloading 1-2MB chunks with generation size $g \in \{12, 20, 24\}$ over one 1,792 KB/s and three 896 KB/s connections with window size $w = 240$.

We found that the generation size does not have significantly impact normalized goodput, as FIGURE 13 shows. This is an important result because we can reduce the generation size and thereby the computation overhead of the system, without having significant performance loss.

d: Normalized goodput for window size $w \in \{120, 240, 360\}$

FIGURE 14 presents normalized goodput for different window sizes. The performance of Parallel HTTP is not affected by the window size we set. In case of small window size, Uncoded MUTP has the same normalized goodput as Parallel HTTP. With the increase of w, Uncoded MUTP performs significantly better, reaching a 40% gain for $w = 480$ compared to the HTTP approach. Coded MUTP has a further up to 16% performance gain compared to Uncoded MUTP regarding normalized goodput.

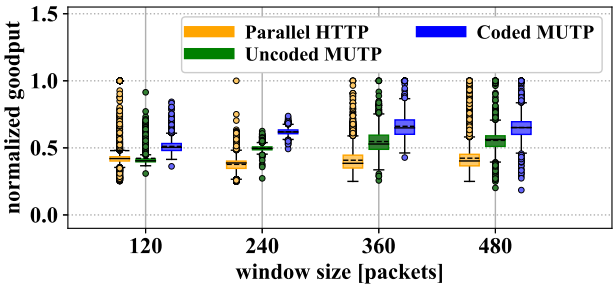

FIGURE 14. Normalized goodput of downloading 1-2MB chunks with window size $w \in \{120, 240, 360\}$ over one 1,792 KB/s and three 896 KB/s connections with generation size $g = 24$.

e: Goodput impact of packet loss

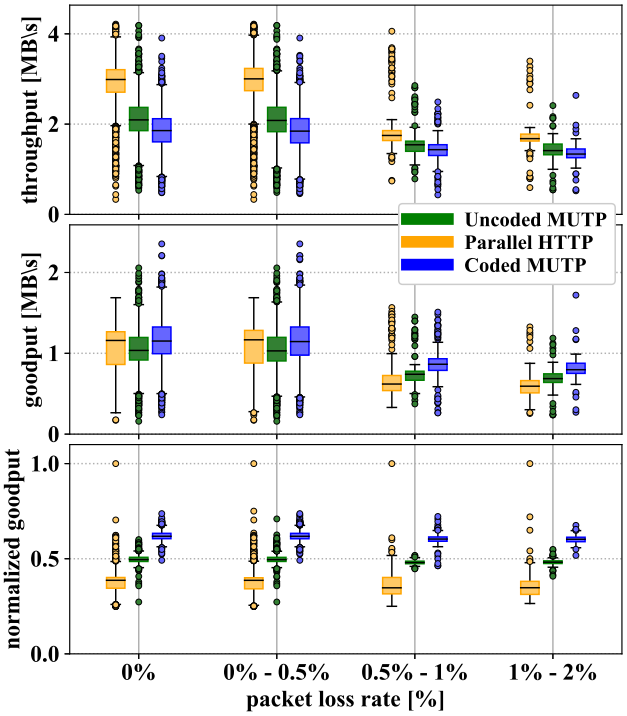

FIGURE 15. Goodput of downloading 1-2MB chunks over lossy link with one 1,792 KB/s and three 896 KB/s connections with window size $w = 240$ and generation size $g = 24$.

Our system is capable of measuring the packet loss rate for our MUTP protocols that work over the WebRTC data channel. Since it is not possible to measure the packet loss rate from JavaScript for an HTTP download, therefore we inferred the loss rate for the Parallel HTTP downloads: For each download, we saved the completion time and also the used proxies and a Universally unique identifier (UUID) that anonymously identifies the downloading client. We calculated the packet loss rate for the HTTP downloads by averaging the loss rate for those downloads that was initialized by the same client with the same proxy servers in the last or following 30 seconds.

FIGURE 15 shows the goodput for different packet loss rates. If the packet loss rate is below 0.5%, the throughput and goodput of the system are constant. There is a significant throughput drop as the loss rate is over 0.5%. The two MUTP protocols only have a small performance decrease, while the throughput of the Parallel HTTP reduces significantly. In the case of the goodput, while the drop rate is below 0.5%, all three download approaches perform similarly. As the loss rate reaches 1%, Parallel HTTP has a significant performance loss compared to the MUTP protocols. The results also show that the higher loss rate does not have a negative influence on the packet scheduling of the MUTP protocols since the normalized good stays constant for all measured loss rates.

f: Comparison to related work

We also compared our system to the Network Coded Filesystem Shim (NCFSS) from Sørensen et al. [18]. They have proposed a filesystem-level solution for multipath, and multi-source download through three approaches: *Naive*, *Chunked* and RLNC *Coded*. Their Naive solution slices the source file into large, equal-sized parts, and the client requests one part from each source. In contrast to that, we focus on a loosely orchestrated scenario where the number of responding servers is not always known in advance. Therefore we use a Parallel HTTP solution, where the same source data is requested from all sources. Furthermore, in the case of the *Naive* approach, some parts may become straggler if some of the sources have significantly lower bandwidth. The Chunked approach slices the source file into small (16-32 KB) chunks, like Uncoded MUTP, but their solution works in a pull fashion by requesting each chunk separately, instead of a push fashion as Uncoded MUTP works. Their RLNC Coded approach applies network coding on the Chunked parts, just as Coded MUTP extends Uncoded MUTP.

Their empirical result shows that the Naive approach outperforms the Chunked solution as Chuked requests each chunk separately that adds significant overhead to the communication. In contrast to that, our Uncoded MUTP outperforms the Parallel HTTP as MUTP can skip some of the feedback that can significantly reduce the communication overhead. Furthermore, Parallel HTTP request the same source data from all sources, and the fastest response is used, instead of requesting different parts from all sources. Their Coded solution outperforms their Naive and Chunked solution, just as Coded MUTP outperforms Parallel HTTP and Uncoded MUTP. On the other hand, their Coded solution has a significantly higher gain over the other two approaches than Coded MUTP has. The gain difference comes from the used technology as JavaScript has a poor performance on carrying out extensive mathematical calculations. They used 10 MB and 100 MB data size, while YouTube chunks tend to be less than 5MB, and results show that on bigger data, network coding has higher throughput. Furthermore, our solution avoids the straggler problem because of the commutative feedback that adds extra overhead to the system.

VI. CONCLUSION

In this paper, we have proposed two multi-source download protocols for loosely orchestrated multi-source network scenarios: MUlti-source Transmission Protocol (MUTP) and Coded MUTP. To test the performance of our protocols, we developed a system that contains browser extensions on the client-side and several proxy servers on the server-side. The browser extensions intercept YouTube video downloads and forward them through multiple proxies by using a simple Parallel HTTP, the Uncoded MUTP or the Coded MUTP approach. We deployed our proxy servers to five different locations on multiple continents using Amazon Web Services. We carried out an extensive measurement campaign that ran for more than eleven months. Our results show that the heavily optimized HTTP protocol outperforms our MUTP protocols when downloading from only a single server. As we increase the number of sources to two, the mean goodput of Coded MUTP matches the mean goodput of the HTTP approach, when downloading 1-2 MB sized data. Further increasing the number of sources to four, both MUTP protocols outperform the simple Parallel HTTP. Throughout our measurements, we achieved two- and three-fold normalized goodput increase with Uncoded MUTP and Coded MUTP, respectively. Our results show that applying Random Linear Network Coding in a loosely orchestrated multi-source scenario can achieve significant goodput increase. Furthermore, we show that our multi-source protocols avoid the straggler problem. Therefore, adding new sources to the network increases the goodput.

As future work, we plan to investigate further packet scheduling methods for both Uncoded and Coded MUTP protocols. In this research, we used WebRTC as the underlying protocol for our MUTP protocols, to make our system widely available and easy to use, as it only required a simple browser extension install. As we presented in our results, WebRTC introduced a significant overhead compared to a UDP connection. We plan to adapt our MUTP protocols to work directly over UDP to have better control over the configuration of the underlying network.

Our work shows the potential of coded multi-source downloads that has high applicability in Information-Centric Networking (ICN) [1] and distributed systems [27].

REFERENCES

[1] S. N. S. Hashemi and A. Bohlooli, "Analytical modeling of multi-source content delivery in information-centric networks," *Computer Networks*, vol. 140, pp. 152 – 162, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618302056

[2] L. Kong, J. Zhu, R. Dai, and M. N. Sadat, "Impact of distributed caching on video streaming quality in information centric networks," in *2017 IEEE International Symposium on Multimedia (ISM)*, Dec 2017, pp. 399–402.

[3] (2018, Nov.) Ericsson mobility report. Ericsson. [Online]. Available: https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf

[4] (2018) Global Internet Phenomena. Sandive. [Online]. Available: https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf

[5] R. Ahmad, E. A. Sundararajan, N. E. Othman, and M. Ismail, "Handover in LTE-advanced wireless networks: state of art and survey of decision algorithm," *Telecommunication Systems*, vol. 66, no. 3, pp. 533–558, Nov 2017. [Online]. Available: https://doi.org/10.1007/s11235-017-0303-6

[6] N. Wehner, S. Wassermann, P. Casas, M. Seufert, and F. Wamser, "Beauty is in the eye of the smartphone holder a data driven analysis of youtube mobile qoe," in *2018 14th International Conference on Network and Service Management (CNSM)*, Nov 2018, pp. 343–347.

[7] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, "Bringing the cloud to the edge," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2014, pp. 346–351.

[8] P. J. Braun, S. Pandi, R. Schmoll, and F. H. P. Fitzek, "On the study and deployment of mobile edge cloud for tactile internet using a 5g gaming application," in *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan 2017, pp. 154–159.

[9] J. Miyoshi, S. Kawauchi, M. Bandai, and M. Yamamoto, "Multi-source congestion control for content centric networks," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ser. ACM-ICN '16. New York, NY, USA: ACM, 2016, pp. 205–206. [Online]. Available: http://doi.acm.org/10.1145/2984356.2985235

[10] Y. Thomas, C. Tsilopoulos, G. Xylomenos, and G. C. Polyzos, "Multisource and multipath file transfers through publish-subscribe internetworking," in *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, ser. ICN '13. New York, NY, USA: ACM, 2013, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2491224.2491238

[11] J. Bruneau-Queyreix, M. Lacaud, D. Négru, J. M. Batalla, and E. Borcoci, "Adding a new dimension to http adaptive streaming through multiple-source capabilities," *IEEE MultiMedia*, vol. 25, no. 3, pp. 65–78, July 2018.

[12] J. M. Batalla, P. Krawiec, C. X. Mavromoustakis, G. Mastorakis, N. Chilamkurti, D. Negru, J. Bruneau-Queyreix, and E. Borcoci, "Efficient Media Streaming with Collaborative Terminals for the Smart City Environment," *IEEE Communications Magazine*, vol. 55, no. 1, pp. 98–104, January 2017.

[13] H. Pucha, D. G. Andersen, and M. Kaminsky, "Exploiting Similarity for Multi-source Downloads Using File Handprints," in *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2–2.

[14] R. Ahlswede, N. Cai, S. Y. Li, and R. W. Yeung, "Network Information Flow," *IEEE Trans. Inf. Theor.*, vol. 46, no. 4, pp. 1204–1216, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/18.850663

[15] P. J. Braun, D. Malak, M. Médard, and M. Ekler, "Multi-Source Coded Downloads," in *2019 IEEE International Conference on Communications (ICC)*, May 2019, pp. 1–7.

[16] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, "Network Coding Meets TCP: Theory and Implementation," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, March 2011.

[17] M. Kim, T. Klein, E. Soljanin, J. a. Barros, and M. Médard, "Modeling network coded tcp: Analysis of throughput and energy cost," *Mob. Netw. Appl.*, vol. 19, no. 6, pp. 790–803, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1007/s11036-014-0556-1

[18] C. W. Sørensen, D. E. Lucani, and M. Médard, "On network coded filesystem shim: Over-the-top multipath multi-source made easy," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–7.

[19] M. A. M. Songze Li and A. S. Avestimehr, "A Unified Coding Framework for Distributed Computing with Straggling Servers," *CoRR*, vol. abs/1609.01690, 2016. [Online]. Available: http://arxiv.org/abs/1609.01690

[20] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 265–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924962

[21] P. J. Braun, D. Malak, M. Médard, and M. Ekler, "Enabling multi-source coded downloads," in *2019 IEEE International Conference on Edge Computing (EDGE)*, Milan, Italy, July 2019.

[22] (2011, Dec.) Websocket. [Online]. Available: https://tools.ietf.org/html/rfc6455

[23] M. V. Pedersen, J. Heide, and F. H. P. Fitzek, "Kodo: An Open and Research Oriented Network Coding Library," in *NETWORKING 2011 Workshops*, V. Casares-Giner, P. Manzoni, and A. Pont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 145–152.

[24] R. L. Xia and J. K. Muppala, "A Survey of Bittorrent Performance," *IEEE Communications Surveys Tutorials*, vol. 12, no. 2, pp. 140–158, Second 2010.

[25] P. J. Braun, M. Sipos, P. Ekler, and F. H. P. Fitzek, "On the Performance Boost for Peer to Peer WebRTC-based Video Streaming with Network Coding," in *2017 IEEE International Conference on Communications (ICC)*, May 2017, pp. 1–6.

[26] S. Taheri, L. A. Beni, A. V. Veidenbaum, A. Nicolau, R. Cammarota, J. Qiu, Q. Lu, and M. R. Haghighat, "Webrtcbench: a benchmark for performance assessment of webrtc implementations," in *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, Oct 2015, pp. 1–7.

[27] N. Anjum, D. Karamshuk, M. Shikh-Bahaei, and N. Sastry, "Survey on peer-assisted content delivery networks," *Computer Networks*, vol. 116, pp. 79 – 95, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128617300464