Power Efficient Multi-CDN Communication over Content Steering Server

Burak Kara, Gwendal Simon

To cite this version:

Burak Kara, Gwendal Simon. Power Efficient Multi-CDN Communication over Content Steering Server. MMSys 2024 - 15th ACM Multimedia Systems Conference, Apr 2024, Bari, Italy. pp.478-484, ⟨10.1145/3625468.3652196⟩. ⟨hal-04571125v2⟩

HAL Id: hal-04571125

https://inria.hal.science/hal-04571125v2

Submitted on 17 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright - All rights reserved

Power Efficient Multi-CDN Communication over Content Steering Server

Burak Kara[†⋆] and Gwendal Simon[†]
[†]Synamedia, [⋆]Inria
Rennes, France

ABSTRACT

The Content Delivery Networks (CDNs) in a multi-CDN delivery have variable requirements and capabilities. Multi-CDN communication over content steering server aims to address this problem in real-time by allowing a CDN to manage its traffic load based on internal metrics via communication with other CDNs over the content steering server while aligning with the standard. The communication process involves operation requests, acceptances, rejections, or negotiations for client offloading/onloading. The paper implements this approach and utilizes power efficiency as the internal CDN metric. We present the implementation details and the possible lifecycles of a communication process. Future work includes the comprehensive evaluation and introduction of learning-based approaches by predicting future traffic load and initiating communication process before the internal metrics catastrophe.

CCS CONCEPTS

• **Information systems → Multimedia streaming**.

KEYWORDS

Content Delivery Network, Multi-CDN, Content Steering, ABR, Green video streaming, Sustainability

1 INTRODUCTION

To deliver video content at scale over the Internet, the service providers commonly employ multiple vendors, each operating independent CDNs, with a focus on key metrics such as high availability, fault tolerance, and minimized latency [1]. To address this growing trend, known as the multi-CDN strategy, several methods have emerged to orchestrate large-scale content distribution. Bentaleb et al. [4] categorizes them as two: client-based and server-based.

In the client-based strategy, the player switches between suitable CDNs or uses multiple CDNs simultaneously in a session. The input of the CDN selection decision, based on a heuristic rule or a learning-based approach, is a set of measures collected from the player such as buffer level, throughput, video bitrate, and latency.

On the contrary, the server-based strategy involves a decision point typically embodied by a central server. The technical communities behind streaming standards — HTTP Live Streaming (HLS) and Dynamic Adaptive Streaming over HTTP (DASH) — have designed a steering system, where every client regularly contacts a central manager, known as content steering server (CSS), to get instructions on which CDN to retrieve the content from [6, 9]. This system enables an in-session switch between CDNs, as implemented by Silhavy et al. [15] for DASH. From the client feedback and the global metrics collected, for example, using Common-Media-Client-Data (CMCD) standard [5], CSS can implement a load balancing strategy to find a trade-off between long-term business objectives and short-term Quality of Experience (QoE) maximization.

The current content steering implementations prioritize user QoE and business agreements between vendors and do not consider the varying CDN requirements (*e.g.*, capacity and power efficiency) over time [4, 10, 14]. For instance, network operators deploy *private CDNs* in their networks to deliver the content of their pay-TV services in priority, and deliver the content of third-party streaming services only if some egress capacity is left. The capacity of a CDN offered to the content steering service changes over time with the fluctuating pay-TV traffic. The second example is the changing power efficiency of a CDN, influenced by the number of clients being served and the number of cache servers. Each cache server exhibits load-dependent fluctuations. As Beloglazov et al. [3] refer, about two-thirds of the consumption under the peak load is attributed to base hardware and software consumption even without traffic load and the power consumption varies in the remaining one-third based on traffic load. Barroso and Hölzle [2] define this ratio as 50%. Figure 1 presents this in the cumulative power consumption of a CDN with five cache servers. Mathew et al. [13] address this per-server inefficiency in a CDN by energy-aware load balancing between cache servers, yet the work lacks the global power efficiency of a CDN in a multi-CDN delivery. Mainly, the total traffic load of a CDN affects power consumption and the other performance evaluation metrics [16].

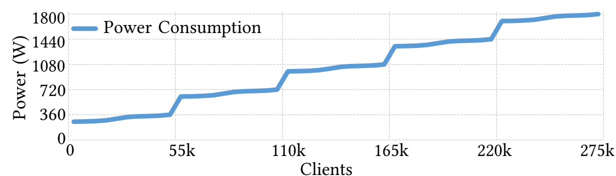

Figure 1: The power consumption jumping with steps at each cache server addition. The jumps indicate the base power consumption of cache servers and the other parts show a logarithmic (yet, near-linear) relation with the increasing number of clients.

Kara and Simon [10] proposed multi-CDN communication over content steering server (mCDNcomm) to enable CDNs to communicate and fine-tune their traffic load through negotiations. This fine-tuning is designed to be achieved by offloading or onloading some portion of traffic (*i.e.*, clients). While aligning with the stateless content steering architecture in [6, 9], the solution claims to enhance internal CDN metrics and address the evolving CDN requirements, varying based on the vendor business model.

This paper demonstrates power efficient content steering by implementing mCDNcomm and leveraging power efficiency as the internal CDN metric. The correlation between power consumption

and traffic load, utilized to calculate the power efficiency, influences the initiation or maintenance of communication between CDNs. Our demonstration illustrates power consumption fluctuations based on the number of clients served by a CDN and explores different scenarios in mCDNcomm affected by dynamic power efficiency considerations. Figure 2 illustrates the motivation of our work, inefficient utilization of the static throughput capacity due to the dynamic egress throughput.

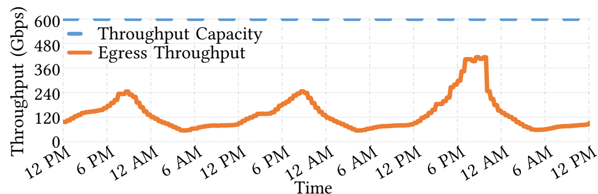

Figure 2: The egress throughput of delivery service (solid – orange) and available throughput capacity (dashed – blue).

In the remainder of the paper, Section 2 presents our work by providing the mCDNcomm implementation details. We describe the demonstration and the setup in Section 3. Section 4 discusses the possible real-life practice of our work and concludes the paper.

2 MULTI-CDN COMMUNICATION

Multi-CDN communication over content steering server (mCDNcomm) enables the communication between CDNs utilizing content steering within a multi-CDN delivery. Specifically, it allows a CDN to tune its traffic load — either offloading a portion or taking on additional load (onloading) — through negotiations with other CDNs. These negotiations exclusively occur over CSS, the only entity with complete awareness of the session. CSS holds the authority to make the final decision on either offloading or onloading traffic for CDNs while ensuring a coordinated and controlled approach to traffic management within the multi-CDN ecosystem. Figure 3 provides an overview of the architecture.

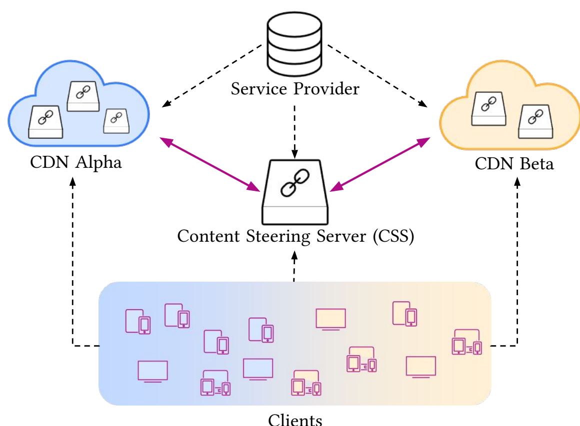

Figure 3: Overview of the content steering architecture extended with mCDNcomm. The purple (solid) lines indicate the communication between elements.

mCDNcomm addresses the dynamic requirements of CDNs while keeping the principles of the content steering standard. These requirements vary based on the internal business models of CDN vendors. Recognizing this diversity, a CSS tailored for each CDN contradicts the stateless approach advocated in the community [6, 9]. Moreover, the reluctance of CDN vendors to disclose internal sensitive metrics poses an additional challenge to the in-session approach, introducing complexities to align evolving CDN needs with the real-time dynamics of content steering. Therefore, as illustrated in Figure 4, the steering controller is integrated as a daemon — plug-in — into both CDNs and CSS to align with the stateless architecture. We employ Python 3.10[1] and Flask 2.3.3[2] to develop steering controller.

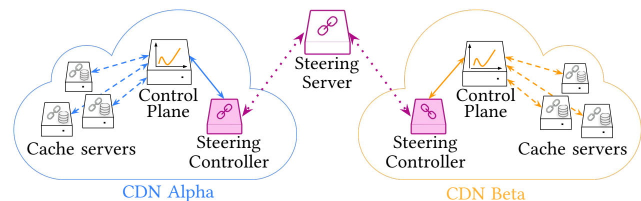

Figure 4: Overview of mCDNcomm, integrating steering controller into CDNs and CSS. The purple (dotted) lines indicate communication paths.

In the steering controller, two main processes work simultaneously: data fetcher and communicator. On a CDN, the data fetcher utilizes Prometheus[3] insights tool at regular five-second intervals. This process collects dynamic metrics exposed by the control plane, including the total CDN capacity, peak traffic power consumption, the number of clients served, and the total power consumption of core CDN components, such as cache servers, control plane servers, traffic router servers, and switches. Then, it informs the communicator with a push-based approach. The control plane possesses a comprehensive awareness of the session by monitoring the resources (*e.g.*, cache servers, traffic routers) and collecting metrics (*e.g.*, egress throughput, capacity) [11]. The communicator decides to initiate and maintain communication (*i.e.*, operation request) with CSS, based on the data received from the data fetcher. It keeps operation records, as well as the lifetime of the current communication process. On CSS, the data fetcher is utilized in a pull-based manner, only when the communicator receives an operation request or needs to evaluate it. The communicator listens to messages from CDNs and keeps records of the operation.

To enhance clarity, we designate the requester CDN as CDN_α. Without loss of generality, the CDNs responding with accept and reject are named CDN_β and CDN_γ, respectively. Similarly, CDN_δ indicates the CDNs that do not respond or respond busy.

2.1 Negotiation Outcomes

In this section, we present the outcomes of a communication process to bring an overview without delving into technical intricacies. The different outcomes are threefold: *completed*, *rejected*, or *aborted.*

[1] https://www.python.org/downloads/release/python-3100/
[2] https://github.com/pallets/flask/releases/tag/2.3.3
[3] https://prometheus.io/

The starting point of each operation request includes two steps: First, the requester (CDN_{α} in our example) initiates an operation by dispatching a request to CSS. CSS assesses and (possibly) modifies this operation request based on some internal data. Then, the three outcomes are issued after a series of steps, as detailed.

Completed Scenario. The conclusion of either a successfully executed operation request or its negotiated counterpart. In a typical *completed* scenario, the refined request is sent to other entities ($CDN_{\beta,\gamma,\delta}$) capable of responding. Upon receiving the offer, at least one of the CDNs, say CDN_{β} in our example, accepts the operation request, either without negotiation or following negotiation on specific terms (*e.g.*, the number of clients).

Rejected Scenario. The conclusion of unsuccessful operation request or its negotiated counterpart. The refined request is sent to the available CDNs. Upon receiving the offer, no CDN accept the operation request, meaning that either $CDN_{\beta,\gamma,\delta}$ reject negotiation terms, or propose negotiated terms that CDN_{α} finds unacceptable.

Aborted Scenario. CSS determines that the operation cannot be fulfilled, either at the current point in time or after evaluating negotiated terms. Additionally, if one or more CDNs, say CDN_{δ} in our example, involved in the communication process become unreachable, CSS aborts the operation request after a timeout.

2.2 Power Efficiency as Internal CDN Metric

We employ the collected metrics to calculate the current and optimal power efficiency, measured in the number of clients per watt. The calculation step involves straightforward mathematical operations. First, the power consumption of the entire CDN (p_{now}) is determined by summing up the collected power consumption numbers. With the number of clients being served (c_{now}), the power efficiency (e_{now} in clients/W) is calculated by dividing c_{now} by p_{now}. Second, for normalization, the best-possible efficiency (e_{target}) is computed by dividing the per-server capacity ($c_{capacity}$) by the peak traffic power consumption (p_{peak}). As the last step, we scale the e_{now} between 0 and 1 (being the worst and the best efficiency, respectively) using e_{target}.

The scaled value (*i.e.*, e_{now}) is utilized to calculate the required number of clients to optimize the power efficiency of the CDN. We present here our implementation but many other processes can be implemented. First, based on the logarithmic increase in the power consumption illustrated in Figure 1, we cluster the number of clients distribution into three ranges: offload, onload, and optimal. The optimal range is defined as the 10% boundary of e_{target}, corresponding to between 1 and 0.9 in the scale. When e_{now} is in this range, we keep the current state of cache servers and do not decide an operation request. The offload range is defined as 0.3, meaning that when a cache server is loaded less than its 30% capacity, the traffic should be offloaded and the cache server should be turned off. For the onload, the remaining range between 0.3 and 0.9 is utilized. Any e_{now} in this range is tried to be promoted to the optimal range by onloading more clients.

During the communication process, CDNs also utilize the optimization logic to simulate the operation request received from CSS. They consider the $simulatedc_{now}$ as $c_{now} \pm count$ where c_{now} is the current number of clients and *count* is the number of clients to be onloaded or offloaded after the operation request. They decide to accept, reject, or negotiate the received operation request by comparing the simulated (*i.e.*, $simulatede_{now}$) and the actual power efficiency (*i.e.*, e_{now}).

2.3 Protocol

This section explores the communication protocol, building upon the insights from the general presentation in Section 2.1.

Step 0. Initially, CDNs and CSS need to be informed about the communication endpoints of each other. In our scenario, this information exchange is facilitated through the streaming manifest (*e.g.*, MPD or m3u8). These manifest files, disseminated by the service provider to CDNs, also contain the address of CSS. Communication endpoints are embedded into the streaming manifest and distributed to all entities involved, ensuring comprehensive awareness.

Step 1. The communication process begins when CDN_{α} determines the need to manage more or fewer clients, considering power efficiency. To calculate the required number of clients to optimize power efficiency as explained in Section 2.2, CDN_{α} generates an operation record (see Code 1) and sends it to CSS. The properties of this record specify the type of the operation as in Table 1, randomly generated unique 128-bit hash to identify the operation requests, the number of clients demanded either offloading or onloading, and operation status as in Table 2, respectively. The first three properties are transmitted to CSS via an Hypertext Transfer Protocol (HTTP) GET request as an operation request. To illustrate this step of the protocol with our example, please refer to Example 1.

```
type: '',    # Operation type defined in Table 1
token: '',   # Unique identifier
count: 0,    # Number of clients
status: null # Operation status defined in Table 2
```

Code 1: The operation record in CDNs.

Let us assume 50K clients in total in the streaming session, distributed between CDN_{α}, CDN_{β}, CDN_{γ}, and CDN_{δ}. CDN_{α} decides to onload additional clients to improve power efficiency since it has the capacity for 20K clients but currently serves only 13K clients. It calculates the required number of extra clients as 7K and generates the operation record, `{ type: 'onload', token: '027eb5...', count: 7000, status: 'offered' }` and sends it to CSS.

Example 1: CDN_{α} **initiates an operation request.**

Step 2. CSS possesses a comprehensive awareness of the streaming sessions, including the total number of clients and their distribution across various CDNs. This knowledge enables CSS to oversee the feasibility of operation request and decide to further transfer it to the other CDNs ($CDN_{\beta,\gamma,\delta}$). After receiving the operation request from CDN_{α}, CSS initializes its operation record as in Code 2. The first three fields of this record are the same as the ones for CDNs. Differently, CSS also keeps the record of the CDNs that participates in the communication process. The *req_cdn* field indicates the id of CDN_{α}. The other fields named *acc_cdns, rej_cdns, neg_cdns, b_cdns* keep the records of *accepted*, *rejected*, *negotiating*, and *busy* CDNs, respectively. If CSS recognizes the impracticality or inconsistency

with current client statistics and/or business arrangements, it may abort the operation request as explained in **Step 2.1**. Otherwise, the communication process continues in **Step 3**.

Table 1: Operation types used in the requests and records.

Type	Description
Offload	Indicate the willingness to serve fewer clients.
Onload	Indicate the desire to serve more clients.

Table 2: Operation request status shared from CSS to CDNs.

Status	Description
Offered	CSS received an operation request from the requester CDN and offered it to other CDN(s).
Accepted	At least one CDN is accepted the operation request to fulfill the operation requirements.
WIP	The operation request is being processed.
Done	The operation request has been fulfilled.
Rejected	The operation request is rejected by the offered CDN(s) and can not be fulfilled.
Aborted	The offered CDN(s) are busy or unavailable. Alternatively, the operation request may not be fulfilled according to the current CSS statistics.

```
type: 'onload',   # Operation type defined in Table 1
token: '027eb5...', # Unique identifier
count: 7000,      # Number of clients
status: null,     # Operation status defined in Table 2
req_cdn: 'alpha', # Requester CDN
acc_cdns: [],     # Accepted CDN(s)
rej_cdns: [],     # Rejected CDN(s)
neg_cdns: [],     # Negotiating CDN(s)
b_cdns: [],       # Busy CDN(s)
```

Code 2: The operation record in CSS.

Step 2.1. CSS decides to abort the operation request before offering to other CDNs. CSS sends *aborted* to CDN_α via HTTP GET request and updates *status* field to *aborted*. After this stage, CSS ceases to accept any additional messages related to this operation request thanks to *status* field. The informed CDN_α also updates its operation record and does not accept further updates for this operation request. Meanwhile, it also puts itself in cooldown mode in which it has to wait for default Time-To-Live (TTL) time before deciding on a new operation. The default TTL value is defined in the service provider and provided to CSS as part of the DASH Content Steering Manifest (DCSM) and HLS Content Steering Manifest (HCSM) [6, 9]. The abortion scenario is described in Example 2.

Step 3. CSS extends the offer to others ($CDN_{\beta,\gamma,\delta}$) and updates the *status* field to *offered*. Simultaneously, CSS informs CDN_α about the status update via an HTTP GET request. CDN_α updates its record in Code 1 by setting *status* field to *offered*, and awaits further updates from CSS. For CDNs, the time threshold to abort the operation request is defined as TTL, indicating that any operation request should be completed within TTL time units after its initialization. The example scenario for this step can be seen in Example 3.

Building on Example 1, after evaluating the operation request, CSS realizes that CDN_α can not have more than 15K clients based on the agreements between CDNs. CSS aborts the operation request, updates its record with `{ status: 'aborted' }` and sends `{ token : '027eb5...', status: 'aborted' }` to CDN_α. With this message, CDN_α updates its record in Code 1 by setting `{ status: 'aborted' }`. The operation lifetime is concluded as *aborted*.

Example 2: CSS aborts and informs CDN_α.

In an alternative scenario in Example 2, this time, CSS evaluates the operation request and finds out that 7K clients can be offloaded to CDN_α from other CDNs. Therefore, it offers the operation request to $CDN_{\beta,\gamma,\delta}$ via HTTP GET request containing `{ type: 'onload', token: '027eb5...', count: 7000 }` while updating its record with `{ status: 'offered' }`. It also informs CDN_α and allows it to update its record with `{ status: 'offered' }`.

Example 3: CSS extends the operation request to other CDNs.

Step 4. The other CDNs receive the operation request from CSS and initialize an operation record following the structure outlined in Code 1. For the unavailable CDNs, the communication process ends in **Step 4.1**. The available ones, *i.e.*, not currently engaged in another operation process, calculate their internal statistics and simulate the offer, similar to the process outlined in Section 2.2. In the acceptance case of the offer, the communication process continues in **Step 4.2**. In the rejection case of the offer, the communication process ends in **Step 4.3**. As the third option, as explained in **Step 4.4**, they can negotiate the offer for their best interest.

Step 4.1. The unavailable CDNs respond with *status* field as *busy* and update their operation record. Upon receiving their responses, CSS records these CDNs to *b_cdns* list. From now on, for these CDNs, the communication process concludes at this step and they receive no further message from CSS. In our work, the status field in the responses returned from both CDNs corresponds to one of the values presented in Table 3. This step is exemplified in Example 4.

CDN_δ in Example 3 is in the cooldown mode due to the previous operation request; therefore, it initializes the operation record `{ token: '027eb5...', status: 'busy' }` and responds with the same record, without evaluating the operation request. For CDN_δ, the process concludes at this step and it receives no further message from CSS. In the meantime, CSS, which just received the *busy* status, updates its record by appending CDN_δ to `b_cdns` list.

Example 4: CDN_δ responds busy.

Step 4.2. Some of the available CDNs decide to accept the operation request, based on their internal decision and simulation results, and update their operation record. They send a response with *status* field as *accepted* to CSS. Upon receiving their responses, CSS records these CDNs to *acc_cdns* list. The communication process continues with the CDNs recorded in this list. Our example for this case is detailed in Example 5.

Step 4.3. Some of the available CDNs may decide to reject the operation request, based on their internal decisions and simulation results. They also update the *status* field in the operation record to

Table 3: Response types used by the CDNs on CSS reply.

Type	Description
Accepted	The operation request is accepted by the CDN and can be proceed.
Negotiating	CDN is negotiating for the number of clients in the operation request.
Preparing	CDN is tuning the internal resources to deploy more or shutdown some cache servers.
Ready	CDN is done with the preparation and ready to proceed with the operation request.
Rejected	The operation request is rejected by CDN.
Busy	CDN is busy to process an operation request.

Continuing Example 3, CDN_β decides to accept the operation request since offloading 7K clients leads a better power efficiency, based on its calculations. CDN_β updates its record and responds to CSS with `{ token: '027eb5...', status: 'accepted' }`. In the meantime, CSS, which just received the *accepted* status, updates its record by appending CDN_β to `acc_cdns` list.

Example 5: CDN_β accepts the operation request.

rejected. Similar to CSS behavior in **Step 2.1**, they cease to accept any additional messages related to this specific operation request thanks to this *status* field. CSS appends these CDNs to *rej_cdns* list, after receiving their responses, similarly in **Step 4.1**. While the CDNs in both *rej_cdns* and *b_cdns* lists do not take place in the further steps of the communication process, keeping a separate record of them helps differentiate CDN behaviors. For instance, if there is an 'always rejecter' CDN (*i.e.*, rejecting every operation request for last 10 minutes), CSS might exclude this CDN from future operation requests. This feature is left as a future work. Our example for this case is detailed in Example 6.

Building on the scenario in Example 3, this time, unlike *accepted* CDN_β in Example 5, CDN_γ decides to reject the offloading of 7K clients since it reduces the power efficiency. It updates its record and responds to CSS with `{ token: '027eb5...', status: 'rejected' }`. CSS appends it to `rej_cdns` list in its operation record.

Example 6: CDN_γ rejects the operation request.

Step 4.4. In this negotiation case, CDNs utilize the calculation results from **Step 4** and anticipate a more efficient state . This leads to the negotiation of terms (*i.e.*, *count*) in the operation request. This case is further detailed in Example 7.

Step 5. CSS receives responses defined in the previous steps and updates its record outlined in Code 2. If it does not receive any response in TTL time duration from a CDN, that CDN is threaded as *busy* (*i.e.*, appended to *b_cdns* list). CSS iterates the negotiating CDN(s), if and only if there is no CDN already accepting the offer. In the negotiation case, the communication process continues in **Step 5.1**. When the offer is accepted by one or more CDN in CDN_β, the communication process continues in **Step 5.2**. When the offer is neither accepted nor negotiated by the CDN(s) and rejected by at least one CDN, the communication process continues in **Step 5.3**.

Continuing the scenario in Example 3, however this time, CDN_β in Example 5 does not accept the operation request instead it negotiates. CDN_β calculates that offloading 7K clients leads to a lower power efficiency. However, it also anticipates that if it could offload 5K clients, the power efficiency would be improved. Based on these calculations, CDN_β updates the fields in its record with `{ token: '027eb5...', count: 5000, status: 'negotiating' }`. It also sends this refined operation request to CSS with the new terms. CSS appends it to `neg_cdns` list in its operation record.

Example 7: CDN_β negotiates the operation request.

Step 5.1. CSS iterates the negotiating offers and calculates the closest possible cumulative number to the *count* value in the original operation request. For the cases where the cumulative number exceeds the *count* in the initial offer, CSS modifies the negotiated offer with the smallest *count* and negotiates with the corresponding CDN. After having equilibrium, CSS offers this cumulative number to CDN_α to be evaluated. CDN_α informs CSS on acceptance, rejection, or negotiation of the modified offer. If accepted, the process is similar to **Step 5.2**, but this time, CSS informs negotiating CDN(s) instead of CDN_α. If rejected, the process is similar to **Step 5.3**, but again, this time, CSS informs negotiating CDN(s). In the negotiation case, similar to **Step 4.4**, this time CDN_α makes a counter offer and communication process continues in **Step 5**. The negotiation message exchange continues until resolved with *acceptance*, *rejection*, or *abortion* in TTL time threshold.

Step 5.2. CSS updates the *status* field as *accepted* and informs CDN_α. Then, the lifetime continues from **Step 6**. In the meantime, CSS does not communicate with CDN(s) which returns busy, rejected, or negotiating.

Step 5.3. CSS updates the *status* field as *rejected* and informs CDN_α. For the next operation request, CDN_α can use this reject information to offer an operation request with more favorable terms. This learning-based approach is left a future work. The operation lifetime is concluded as *rejected*.

Step 6. CSS sends start message to CDN_α and the first CDN in the *acc_cdns* list, hereby also referenced as CDN_β, as illustrated in Example 8. When CDN_β encounters any problem and becomes unavailable, the communication process described after this step reverts to this state and repeats the protocol with the next CDN in *acc_cdns*. This ensures that the communication process can adapt to changes or issues, allowing for robustness and continued operation even in the presence of disruptions.

Step 7. After receiving the start message from CSS, CDN_α and CDN_β start preparations. First, they update the *status* field: as *preparing* and inform CSS. The preparation steps include deploying more, scheduling shutdown, waking up from sleep mode, or putting into sleep mode for the cache server, depending on the operation type. Once the preparations are done, CDN_α and CDN_β send *status: 'ready'* to inform CSS. This step is exemplified in Example 9.

Continuing with the *accepted* scenario in Example 5, after receiving the responses from CDN_β, CDN_γ, and CDN_δ or waiting TTL time, CSS sends `{ token: '027eb5...', message: 'start' }` to CDN_α and CDN_β based on the operation record. In this example, the record is as follows: `{ type: 'onload', token: '027eb5...', count: 7000, status: 'accepted', req_cdn: 'alpha', acc_cdns: ['beta'], rej_cdns: ['gamma'], neg_cdns: [], b_cdns: ['delta'] }`

Example 8: CSS sends *start* message to CDN_α and CDN_β.

In the ongoing example scenario, CDN_α and CDN_β start preparations after receiving the start message from CSS. Since the operation is *onload* for CDN_α, it may need to run more control-plane servers, horizontally scale its cache servers, or wake them up from sleep mode. Unlike CDN_α, CDN_β may need to reduce its resources since the operation is *offload* from its perspective. Once the preparations are done, CDN_α and CDN_β send `{ token: '027eb5...', status: 'ready' }` to inform CSS.

Example 9: CDN_α and CDN_β start preparations and send ready message.

Step 8. Upon receiving the *ready* message from CDN_α and CDN_β, CSS updates the record with *status: 'WIP'*. It also informs these CDNs with the status. During *WIP* state, CSS puts this new load-balancing decision into action by updating the *PATHWAY-PRIORITY* list in the content streaming manifest (*i.e.*, DCSM or HCSM). This enforces *count* clients (*i.e.*, number of clients in the operation record) to switch CDN. Once the number of clients in the operation request and the number of clients who return the modified content steering manifest is met, the operation is concluded as *completed*. CSS updates the record with *status: 'Done'* and informs participating CDNs (*i.e.*, CDN_α and CDN_β). The participating CDNs also update their records with the *done* status. After this stage, these CDNs and CSS do not accept any further message or update for this operation record. Example 10 concludes our scenario started at Example 1.

After receiving the *ready* from Example 9, CSS starts manipulating the content streaming manifest for clients served with `{ PATHWAY-PRIORITY: ['beta', 'alpha', 'gamma', 'delta'] }`. Instead, CSS serves the manipulated list in the manifest as `{ PATHWAY-PRIORITY: ['alpha', 'beta', 'gamma', 'delta'] }`. Once CSS modify the manifest for 7K clients, the communication process is concluded. CSS sends `{ token: '027eb5...', status: 'done' }` to CDN_α and CDN_β as the final message in the communication process. The whole protocol leads ≈7K clients to switch from CDN_β to CDN_α.

Example 10: The operation request, thus the communication process, is concluded.

3 DEMO DESCRIPTION AND SETUP

In our demonstration, we play a test video using the reference DASH player[4], sampling the latest content steering standard. On the same page, we present the real-time internal CDN metrics (*i.e.*, power efficiency, number of clients being served) and incoming and outgoing messages during the mCDNcomm.

The demonstration setup involves one CSS, one player, and two CDNs (CDN_α, CDN_β) with five and one cache servers. We have three configurations to demonstrate various mCDNcomm scenarios as the response behavior of CDN_β. It can be either *accepter*, *rejecter*, and *natural*, meaning that accepts every incoming operation request, rejects every incoming operation request, or acts naturally and decides based on the internal metrics, respectively. On the other hand, CDN_α acts naturally in the operation decision.

4 DISCUSSION AND CONCLUSION

The prioritization of power efficiency methods may diverge from the practical needs of contemporary businesses, where profit often takes precedence over environmental concerns. Studies like the one presented in this paper, which prioritize power efficiency to reduce the environmental impact of video delivery, may be considered optimistic in light of current business realities. However, as highlighted in previous papers [7, 8, 11, 12], impending regulations aligned with sustainable development goals could make such studies indispensable in the market. These regulations might compel vendors in video delivery, including video service providers, CDN vendors, and infrastructure providers, to meet specific environmental goals, such as achieving a certain percentage of power efficiency or utilizing green energy sources to a designated extent. Furthermore, with the agile load-balancing nature of our work, CDN vendors and their costumers can have more dynamic plans instead of rigid long-term commitments.

The future work includes a comprehensive evaluation of the power consumption and a better integration of the cost related to repeated cache server turn-off and turn-on, including hardware wear. We plan to present the total and individual power consumption of CDNs in two multi-CDN delivery services, one service employing mCDNcomm and the other one not. Furthermore, extending this approach with learning-based methods to predict the fluctuating number of clients will enable more power-efficient resource scaling in video delivery.

[4]https://reference.dashif.org/dash.js/nightly/samples/advanced/content-steering.html

REFERENCES

[1] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. 2012. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *IEEE INFOCOM*. https://doi.org/10.1109/INFCOM.2012.6195531

[2] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (2007), 33–37.

[3] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems* 28, 5 (2012). https://doi.org/10.1016/j.future.2011.04.017

[4] Abdelhak Bentaleb, Reza Farahani, Farzad Tashtarian, Hermann Hellwagner, and Roger Zimmermann. 2023. Which CDN to Download From? A Client and Server Strategies. In *ACM MHV*. https://doi.org/10.1145/3588444.3591030

[5] Consumer Technology Association. 2020. Web Application Video Ecosystem – Common Media Client Data (CMCD). [Online] Available: https://shop.cta.tech/collections/standards/products/web-application-video-ecosystem-common-media-client-data-cta-5004. Accessed on Apr. 22, 2023.

[6] DASH-IF. [n. d.]. DASH-IF Candidate Technical Specification: Content Steering for DASH. [Online] Available: https://github.com/Dash-Industry-Forum/Dash-Industry-Forum.github.io/files/11722876/DASH-IF-CTS-001-1.0.0.pdf. Accessed on Jan. 19, 2024.

[7] European Union. [n. d.]. Rolling Plan for ICT standardisation. [Online] Available: https://joinup.ec.europa.eu/collection/rolling-plan-ict-standardisation/rolling-plan-2023. Accessed on Oct. 6, 2023.

[8] Greening of Streaming. [n. d.]. Summary of our June 21st EU Parliamentary Interface and LESS Accord Brainstorm. [Online] Available: https://www.greeningofstreaming.org/post/summary-of-our-june-21st-eu-parliamentary-interface-and-less-accord-brainstorm. Accessed on Oct. 6, 2023.

[9] HLS. [n. d.]. HLS Content Steering Specification. [Online] Available: https://developer.apple.com/streaming/HLSContentSteeringSpecification.pdf. Accessed on Jan. 19, 2024.

[10] Burak Kara and Gwendal Simon. 2024. Dynamic Content Steering Services: How to Make Everyone Happy in a Multi-CDN World. In *ACM Mile-High Video Conference (MHV '24)*. https://doi.org/10.1145/3638036.3640252

[11] Burak Kara, Gwendal Simon, Bruno Tuffin, Jerome Vieron, and Ali C. Begen. 2023. Studying Green Video Distribution as a Whole. In *ACM Proceedings of the First International Workshop on Green Multimedia Systems*. https://doi.org/10.1145/3593908.3593944

[12] Reinhard Madlener, Siamak Sheykhha, and Wolfgang Briglauer. 2022. The electricity- and CO2-saving potentials offered by regulation of European video-streaming services. *Energy Policy* 161 (2022), 112716.

[13] Vimal Mathew, Ramesh K Sitaraman, and Prashant Shenoy. 2012. Energy-aware load balancing in content delivery networks. In *2012 Proceedings IEEE INFOCOM*. IEEE, 954–962.

[14] Robert Seeliger, Stefan Pham, and Stefan Arbanowski. 2023. End-to-End Optimizations for Green Streaming. In *ACM Proceedings of the First International Workshop on Green Multimedia Systems*. https://doi.org/10.1145/3593908.3593945

[15] Daniel Silhavy, Will Law, Stefan Pham, Ali C. Begen, Alex Giladi, and Alex Balk. 2023. Dynamic CDN Switching - DASH-IF Content Steering in Dash.Js. In *Proceedings of the 2nd Mile-High Video Conference* (Denver, CO, USA) *(MHV '23)*. Association for Computing Machinery, New York, NY, USA, 130–131. https://doi.org/10.1145/3588444.3591027

[16] Saif ul Islam and Jean-Marc Pierson. 2012. Evaluating energy consumption in CDN servers. In *ICT as Key Technology against Global Warming: Second International Conference, ICT-GLOW 2012, Vienna, Austria, September 6, 2012. Proceedings 2*. Springer, 64–78.