CRDT Web Caching: Enabling Distributed Writes and Fast Cache Consistency for REST APIs

Markus Sosnowski, Richard von Seck, Florian Wiedner, and Georg Carle
Technical University of Munich, Germany
{sosnowski, seck, wiedner, carle}@net.in.tum.de

***Abstract*—Web Application developers have two main options to improve the performance of their REST APIs using Content Delivery Network (CDN) caches: define a Time to Live (TTL) or actively invalidate content. However, TTL-based caching is unsuited for the dynamic data exchanged via REST APIs, and neither can speed up write requests. Performance is important, as client latency directly impacts revenue, and a system's scalability is determined by its achievable throughput. A new type of Web proxy that acts as an information broker for the underlying data rather than working on the level of HTTP requests presents new possibilities for enhancing REST APIs. Existing Conflict-free Replicated Data Type (CRDT) semantics and standards like JSON:API can serve as a basis for such a broker. We propose CRDT Web Caching (CWC) as a novel method for distributing application data in a network of Web proxies, enabling origins to automatically update outdated cached content and proxies to respond directly to write requests. We compared simple forwarding, TTL-based caching, invalidation-based caching, and CWC in a simulated CDN deployment. Our results show that TTL-based caching can achieve the best performance, but the long inconsistency window makes it unsuitable for dynamic REST APIs. CWC outperforms invalidation-based caching in terms of throughput and latency due to a higher cache-hit ratio, and it is the only option that can accelerate write requests. However, under high system load, increased performance may lead to higher latency for non-acceleratable requests due to the additional synchronization. CWC allows developers to significantly increase REST API performance above the current state-of-the-art.**

***Index Terms*—CRDT Web Caching, REST APIs, CDNs, Web Caching, Web Proxies, CRDTs, Network Performance Simulation**

I. Introduction

Web caching is a well-elaborated topic (*e.g.*, [1–3]), and caching dynamic data presents a known challenge [4]. The growing use of Content Delivery Networks (CDNs), the separation of static code and dynamic application data in software development, and the adoption of standardized API formats like Representational State Transfer (REST) [5] or JSON:API [6] have changed how Web applications are built. Conflict-free Replicated Data Types (CRDTs) offer a unique method for building distributed applications and have shown significant progress in recent years (*e.g.*, [7]). These changes and new CRDT developments open up new opportunities to enhance Web application performance. This is relevant because sometimes a server cannot fulfill performance requirements regarding the scalability and latency of an application. A low client latency can directly impact user experience and create loss in revenue [8]. While Web caching promises improved performance, applying it to dynamic REST APIs can be challenging. Current options mainly fall into two categories: caching based on a Time to Live (TTL) or active invalidation of content. Both approaches have limitations: TTL-based caching is unsuited for frequently changing Application Programming Interface (API) data, invalidation-based caching requires CDN-specific logic, neither approach allows a CDN to independently apply changes to the data to improve the latency of write requests, and related work has suggested that push-based approaches can outperform both [2].

This work presents CRDT Web Caching (CWC), a method allowing an origin to keep the cache state of a CDN's Web proxies consistent and enabling CDNs to directly apply mutations to the data. The aim is to provide developers with a new option that combines the simplicity of Hypertext Transfer Protocol (HTTP) caching with the power of a fully distributed system seamlessly integrating into existing applications.

Our key contributions are:

i) definition of CWC and the requirements for enabling it for REST APIs on three different levels;
ii) empirical comparison of different REST API caching strategies based on a simulated CDN deployment; and
iii) published data, scripts, and code [9].

II. Background

We focus on client-server applications, where the client provides a User Interface (UI), and the server manages the main application logic. This approach is commonly used in enterprise applications and typically involves three primary layers: *presentation*, *domain*, and *data source* [10], which are typically realized by a client-side UI, a backend server, and a database, respectively.

In modern applications, the interface between the presentation and the domain layer is often implemented using REST APIs. REST [5] is a popular design paradigm for APIs in a distributed client-server system, covering principles and constraints such as statelessness, cacheability, a uniform interface, and support for multi-layered systems. However, communication with backend servers becomes a bottleneck if users frequently wait for server responses. Ultimately, the physical distance those API calls must traverse imposes a hard limit on application performance. To address this limit, web caching can be utilized, allowing web proxies to answer some requests directly. CDNs operate such proxies worldwide and offer them as service to application providers. If using a CDN, the backend is called *origin*. Web caching is typically realized

978-3-903176-66-9

with HTTP caching (*cf.*, [11]), a TTL-based approach. A more concrete format for REST APIs is JSON:API [6], which defines how resources and relations are represented, accessed, and mutated. JSON:API indicates how REST is implemented in practice: using HTTP to access API endpoints and transmitting data via JSON, revealing REST commonly relies on HTTP caching for its cacheability. Modern CDNs go beyond standard HTTP caching functionality by extending the "expiration-based caching model and additionally expose (non-standardized) interfaces for asynchronous cache invalidation" [12]. For instance, Fastly claims to purge all global caches in less than 200 ms using a bimodal multicast algorithm [13]. Furthermore, Web performance can be improved by servers actively pushing data to caches, as discussed by [2]. In summary, three general Web caching approaches exist: TTL-based, Invalidation-based, and Push-based caching.

In this work, we explore CRDTs for Web caching. CRDTs are used in distributed computing to achieve eventual consistency only through the definition of the data type. Each peer can modify a CRDT without coordinating with other peers. It might result in different peers having different states, but the logic of the CRDT ensures consistency when all changes have been propagated to all peers. Developing powerful CRDTs for actual Web applications is an active field of research, and effective CRDTs for arbitrary JSON data types exist, *e.g.*, [7]. JSON CRDT libraries like Automerge [14] go beyond the basic functionalities and provide mechanisms to synchronize them between peers or a server and its clients.

III. Related Work

Ever since the development of the World Wide Web, there have been numerous efforts to enhance its performance. Especially Web caching of static content (documents, texts, videos, audio, etc.) is a well-elaborated topic.

In 1995, Abrams *et al.* [1] discussed the general potential of Web proxy caching covering HTTP. Bestavros [2] highlighted the performance benefits of servers pushing data to proxies closer to the client based on local popularity and emphasized the need for actively invalidating cached content. In 1997, Baentsch *et al.* [15] refined the concept and demonstrated the performance advantages of replication over caching. Iyengar and Challenger [4] explored caching dynamic data on proxies and suggested an invalidation-based approach. Cao and Liu [3] compared approaches such as TTL, "polling-every-time", and invalidation, concluding that an invalidation-based protocol provides the best cache consistency. However, our results indicate that replication still outperforms invalidation.

Over time, the Internet landscape has evolved, with caching primarily carried out by end devices (*e.g.*, a browser) or CDNs specifically configured by application developers. Ninan *et al.* [16] argued that existing cache consistency mechanisms are unsuitable for CDNs. They propose a concept of cooperative leases where the origin notifies the CDN of any changes and sends an invalidation or updated object. Recently, Abolhassani *et al.* [17] showed that caching strategies on Web proxies remain relevant, interpreting push- and pull-based caching approaches as a cost optimization problem and proposing a combined approach. Wingerath *et al.* [18] developed a different approach to handling dynamic data, suggesting to push application-specific caching logic to the client. Other works explored strategies to populate a cache before a user accesses it. For instance, Wan *et al.* [19] proposed grouping users based on their navigational patterns and pre-fetching web requests according to a common user profile.

To the best of our knowledge, CRDTs have not been considered for Web caching yet nor has an approach similar to CWC been proposed.

IV. CRDT Web Caching (CWC)

We define *CWC* as an approach for distributing application data using CRDTs from a central origin to one or multiple trusted Web proxies with the option for proxies to apply changes to the CRDTs directly. The constantly synchronized CRDTs enable the origin to push changes on the data source to Web proxies to keep the cache states consistent. If the origin defines a set of allowed mutations, the Web proxies can directly apply these mutations via uncoordinated distributed writes to speed up simple write queries.

Web proxies can either grant clients direct access to the CRDTs or provide an automatically generated interface to access the underlying CRDT data. Proxies can only directly modify a CRDT if the contained data is structured according to a well-defined format, *e.g.*, using JSON:API [6], and when simple Create Read Update Delete (CRUD) operations are sufficient. The eventually-consistent and conflict-free nature of CRDTs allows Web proxies to asynchronously forward applied mutations to the origin. CWC facilitates only on-demand replication, *e.g.*, when a client accesses an API endpoint. Any CRDT replica can be removed from a Web proxy once all changes have been successfully forwarded to the origin. Hence, CRDT Web caching combines pull- and push-based caching elements, allowing clients to initiate replication while the origin actively populates relevant caches with updates.

We propose implementing CWC for REST by upgrading the API access to a JSON CRDT [7]. During the upgrade process, the origin provides meta-data informing proxies about the structure of the API, access permissions, allowed mutations, and other details. Web proxies will handle the upgrade transparently to the client by returning regular JSON and providing access to the CRDT only through an automatically generated REST interface. This upgrade mechanism enables applications to selectively use CWC for specific endpoints and seamlessly integrate it into existing REST APIs.

An example deployment illustrating CWC is shown in Fig. 1. It contains a client making a `GET` request to a REST API endpoint. As the API endpoint is distributed as a CRDT, proxy 1 can directly return the current state of the CRDT to the client. Proxy 1 does not replicate the CRDT directly from the origin but via proxy 2. For read-only APIs, changes to the CRDT propagate only from the origin to the proxies. However, when client 2 requests a mutation with the `POST` request, proxy 3 can immediately apply it on the CRDT, provide a

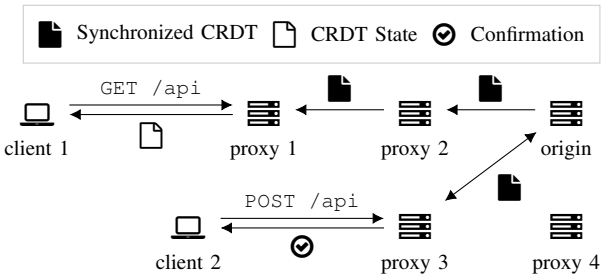


Figure 1. Example deployment utilizing CWC.

confirmation, and synchronize the changes back to the origin later. Since no client accesses the API over proxy 4, it holds no replica of the CRDT.

A. Requirements

CWC for REST APIs can support different levels: basic, advanced, and mutable. The basic level is for read-only data, supports various CRDT definitions (*e.g.*, not limited to JSON CRDTs), and has the following requirements:

R1 A Web proxy stores a mapping of REST API endpoints (*i.e.*, URL patterns) to local CRDT replicas. If the origin upgrades an API request to a CRDT, the proxy creates a local replica of the CRDT and maps the observed endpoint to it. Multiple endpoints can map to the same CRDT. A background task on the proxy ensures constant synchronization of all local CRDTs with the origin.

R2 Received requests to an unknown endpoint are forwarded to the origin. Any read-request (`HTTP GET`) to a known endpoint previously upgraded to a CRDT is answered with the current state of the respective CRDT.

R3 Web proxies periodically forward requests to a known CRDT endpoint to the origin to repeat the upgrade process and adjust for changes in the API metadata. If the connection between the origin and the Web proxy breaks, the Web proxy immediately repeats the upgrade.

R4 The origin informs Web proxies about existing permissions during the upgrade, *e.g.*, which users are allowed to access the data. Web proxies enforce the permissions on all client requests.

Advanced CWC enables a Web proxy to provide clients with additional functionality for read-only API endpoints. In addition to requirements **R1** to **R4**, it requires:

R5 The origin structures the data of the CRDT according to a standardized format that the Web proxy understands, *e.g.*, based on linked data with a well-defined format like JSON:API [6].

R6 The origin informs the Web proxy about the structure of the API, different access patterns, and whether only subsets of the CRDT should be served to clients for some of the patterns. The Web proxy will store the patterns in the local mapping. This enables Web proxies to directly answer a group of API endpoints, *e.g.*, each API call that returns a single object of a collection.

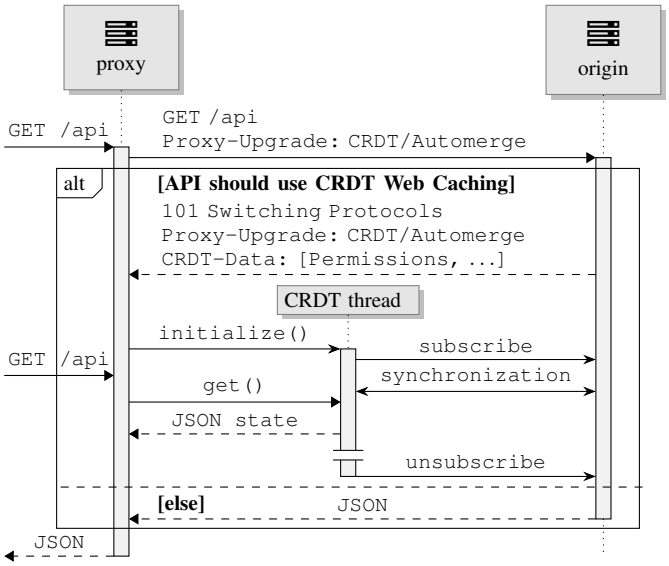


Figure 2. Sequence diagram of an example REST API call where the origin has the choice to use CWC. Future requests can directly start at the second `GET` request, skipping the initial round-trip to the origin.

Mutable CWC enables proxies to directly change the data and requires, in addition to **R1**-**R6**:

R7 The origin flags a CRDT as mutable and informs the Web proxy which HTTP methods (`GET`, `POST`, `DELETE`, etc.) and which operations are allowed. The origin upgrades all allowed methods to the same CRDT. The Web proxy stores the corresponding methods in the local mapping.

R8 A Web proxy flags local CRDT replicas as *dirty* if it applies a mutation. The flag is removed when all changes are successfully synchronized with the origin. No dirty replicas are purged from the local cache of a proxy.

B. Upgrading a REST API Endpoint to Use CWC

An example for an upgrade to a JSON CRDT is shown in Fig. 2. The process starts with a client accessing a REST API endpoint that is expected to return a JSON object. A Web proxy receives the request and forwards it to the origin according to standard HTTP proxy rules. The Web proxy appends its support for CRDTs in an HTTP header, indicating its support for Automerge [14] by adding `Proxy-Upgrade: CRDT/Automerge`. The origin server then decides whether the endpoint should be upgraded. If not, it responds with a regular HTTP response, and the Web proxy acts as a shared HTTP cache. However, if the endpoint should be upgraded to a CRDT, the origin responds with a `101 Switching Protocols` response and the necessary details to access the CRDT. The origin informs the Web proxy about the permission model, such as a required user authentication, endpoint access-control, and CRDT mutability. Then, the proxy can initialize its local replica of the CRDT and establish synchronization. The local replica will be continuously synchronized in a background thread until the proxy decides to evict it. Once the CRDT instance is fully replicated on the proxy, the current

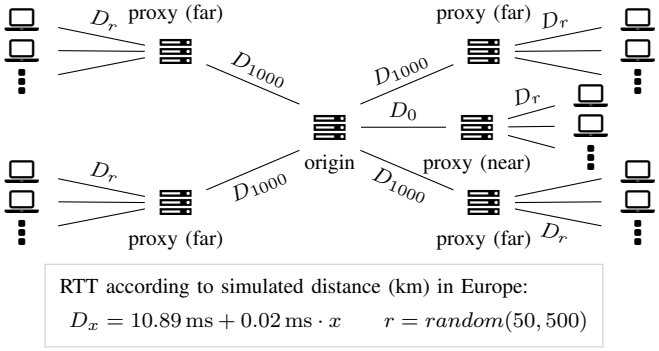


Figure 3. Simulated CDN deployment used to evaluate API caching strategies.

state can be returned to the client as a JSON object. If a client requests the same API endpoint again, the proxy can directly respond with the state of the local replica, skipping the initial round-trip to the origin.

V. Experiment Methodology

To evaluate the performance implications of applying CWC to REST APIs, we simulated a simple CDN deployment with two example applications: a flight booking service and a discussion forum. These applications were chosen to represent different use cases. The flight scenario represents write-heavy applications with complex logic that only the origin can handle. In contrast, the forum scenario involves many reads, and only simple CRUD operations. We tested the four caching strategies using nocache, TTLs, invalidations, and CWC.

A. Experiment Model and Setup

For our study, we modeled a simplified CDN deployment with Mininet [20], a tool capable of simulating complex network typologies on a single machine. The deployment comprises multiple load generators, a few proxies, and a single origin spread evenly over a large area, as illustrated in Fig. 3. The load generators simulate clients; however, they create higher loads than real-world clients because they constantly make requests without the idle times caused by a human user.

We simulated the physical distance covered by network connections with artificial delays. The load generators send their requests to the nearest proxy, which either responds to the requests or forwards them to the origin. For the distances to be realistic, we modeled them after the European Union, approximately $4\,000\,\text{km}$ from west to east and north to south. Martinez *et al.* [21] measured network delays worldwide depending on distance and created a regression model to estimate the RTT on different continents based on distance. For Europe, they measured the function D_x from Fig. 3, allowing us to simulate physical distances in our experiments. The origin is located at the center to best serve all clients. A single proxy is also located at the center, and four more are evenly distributed around the area, each $1\,000\,\text{km}$ away from the origin. Up to 100 load generators connect evenly to the five proxies, each being randomly located between $50\,\text{km}$ and $500\,\text{km}$ from the proxy. The resulting latencies are close to real-world values: [22] measured a median latency of $14\,\text{ms}$ targeting CDNs and $34\,\text{ms}$ targeting data centers in Europe.

B. Evaluated API Caching Strategies

In this work, we compare four different caching strategies:

Nocache The proxies did not cache any responses and only acted as a relay between client and origin.

TTL The APIs utilized an expiration-based strategy where responses to read requests were cached for a defined period. For this study, we choose a TTL of $5\,\text{min}$ as representative value for TTL-based caching. Real-world applications could use lower or higher TTL values depending on their concrete requirements.

Invalidation Responses to read requests were cached with a TTL longer than the experiment. When the origin updates the data source, affected cached responses are invalidated. We implemented this functionality by tagging each response of the origin with one or more keys. Whenever the origin returned a response after changing data, the keys to be invalidated were attached. A proxy observing invalidation keys purged associated local entries and forwarded the list to the other proxies.

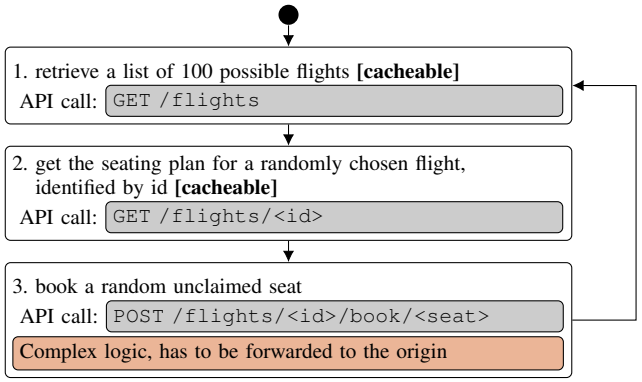
CWC The APIs used advanced and mutable CWC according to Section IV, implemented with Automerge [14]. Our proxies automatically derived REST APIs with the metadata provided during the upgrade. The origin forwarded changes on data objects to all affected CRDTs, effectively updating cached objects on the proxies through Automerge's inherent synchronization protocol.

C. Example Scenarios

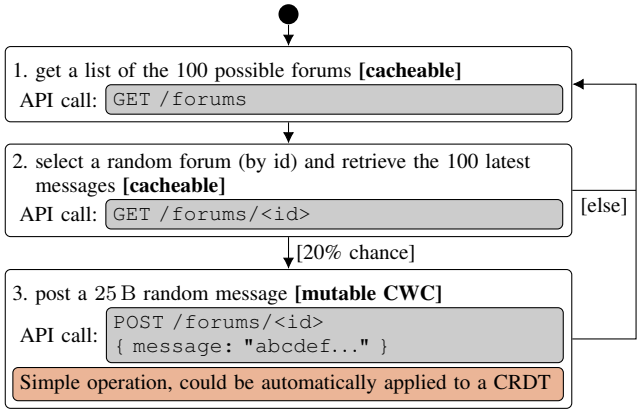
The experiments are initiated load generators acting as clients according to the state machines in Fig. 4. As soon as a client finished the last action of the scenario, it immediately repeated the state machine, as if it was a new client.

In the flight scenario, clients attempt to book a seat on a flight by first requesting a list of 100 possible flights. The client then randomly selects one flight and requests a list of 100 bookable seats for that flight. Each seat has an attribute indicating its availability. If an API caching strategy is being evaluated, both lists are cached. Subsequently, a client attempts to book a random available seat. The booking transaction is assumed to be complex and handled solely by the origin. If a flight is fully booked, the origin removes it from the list of possible flights and creates a new empty one. While caching can enhance performance, it is important for a client to have up-to-date information to avoid attempting to book an already taken seat. The scenario is write-heavy due to each client attempting to book a seat.

In the forum scenario, clients initially requests the list of 100 possible forums, then selects a random forum and retrieve the latest 100 messages. Both read requests are cached if an API caching strategy is evaluated. Then, there is a 20% chance that a client posts a new random message. Mutable CWC is used for the post because adding a new entry to an existing list is a simple CRUD operation. The main challenge of this



(a) Write-heavy Flight Scenario (clients booking a seat on a random flight)



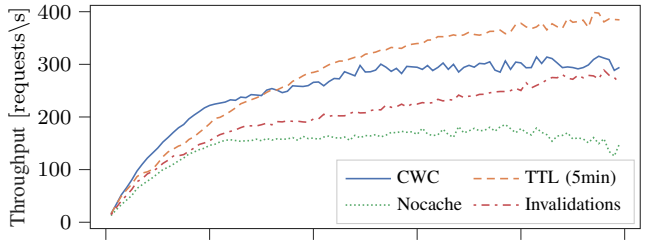
(b) Read-heavy Forum Scenario (clients read and some post messages)

Figure 4. The example scenarios we use to compare the API caching strategies described as state machines.

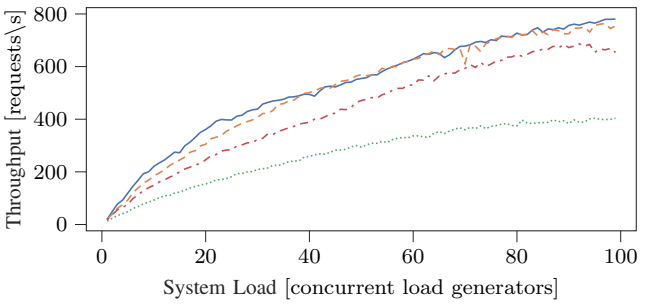
scenario is to enable real-time communication between clients, ensuring they always see the latest messages.

D. Evaluated Metrics

Both latency and throughput are important for client-server applications. In our experiment, we measured throughput as the number of HTTP requests served per second and latency as the time it takes for the client to send an API request and receive a response. To better compare the strategies, we aggregated each request's latency per time frame and differentiated between read and write requests. Incorporating Web proxies into a client-server deployment can improve performance by increasing throughput and reducing latency, but it can lead to consistency issues. To analyze these effects, we need additional metrics such as the *staleness ratio*, *cache-hit ratio*, and the *client-observable inconsistency window*. The staleness ratio indicates the portion of requests answered with outdated data, which can occur even with no caching if content is updated during transmission. The cache-hit ratio reveals the portion of requests that a Web proxy could directly serve from its cache. Both are widely used metrics and have been used already in 1997 by [15]. A newer metric, the client-observable inconsistency window, measures the "time between the commit timestamp and the latest possible read of the previous version for systems that do not expose dirty reads" [23].



(a) Flight Scenario



(b) Forum Scenario

Figure 5. Successful requests per second over time. Every 1 min a new load generator was started, increasing the system load. Subplots share their x-axis.

VI. Evaluation

To evaluate CWC, we used a simple CDN deployment simulated on a single server equipped with two *AMD EPYC 7601 32-Core* processors (128 logical cores) and 1 TB of RAM and we pinned the origin, Web proxies, and clients to separate cores to minimize a CPU scheduling bias. In the experiments, we started with a single load generator and then added a new one every minute up to 100. The load generators continuously made client requests according to the specified scenario. We ensured that there were no cache evictions due to full storage. Additionally, we used an in-memory dictionary as the data source to simulate fast computation. If we can demonstrate the benefits of using a caching approach for such a basic origin server, we can infer that real-world applications with longer processing times (due to complex logic, dedicated databases, etc.) benefit even more.

A. Application Performance: Throughput and Latency

A general metric that reveals the scalability of a system is throughput. A higher value means more clients can be served at the same time. We measured the number of HTTP requests completed per second for both scenarios in Fig. 5. The results reveal that CWC and TTL-based approaches provide the best throughput, followed by an invalidation-based approach. Interestingly, for a few concurrent clients, CWC outperformed the throughput of TTL-based caching by 19% for 20 concurrent clients in the flight scenario, as the content was actively pushed to proxies, resulting in faster request handling. However, the CRDT synchronization on the origin became a bottleneck under higher system load, allowing TTL-based caching to surpass CWC. Despite this, CWC's throughput never dropped below that of invalidation-based

Table I
SUMMARY OF THE SIMULATION RESULTS

Scenario	Caching Strategy	req/s	mean read lat.	mean write lat.	mean inc. window	staleness ratio	cache-hit ratio
Flights	CRDT	252.3	91ms	404ms	701ms	6.5%	97.5%
	Invalid.	200.7	107ms	215ms	1 194ms	60.8%	79.5%
	TTL	281.1	79ms	213ms	5min	98.3%	99.3%
	Nocache	148.6	335ms	331ms	3ms	6.0%	0.0%
Forums	CRDT	527.3	95ms	190ms	1 949ms	26.0%	100.0%
	Invalid.	437.7	106ms	326ms	3 685ms	16.8%	81.2%
	TTL	512.1	85ms	306ms	5min	49.3%	99.6%
	Nocache	270.0	185ms	371ms	1ms	0.6%	0.0%

Note: Abbreviated are requests per second, read/write latency, and client-observable inconsistency window.

caching. Performance is poorest without caching. Generally, the flight scenario has a lower throughput than the forum scenario because it is more write-heavy. CWC challenges the throughput possible with TTL-based approaches while providing the advantage of fast cache consistency.

The latency of read and write requests over time offers an additional perspective on the application's performance. Fig. 6 displays the median read and write latency per minute. It is evident that all three API caching approaches result in similar performance enhancements for read requests compared to no caching, and the increasing load over time has minimal impact. In the absence of caching, the origin becomes the performance bottleneck, leading to a noticeable degradation in latency, particularly in the write-heavy flight scenario. A very different picture is revealed when analyzing the median write latency. In the flight scenario, the origin handles every write; consequently, performance is directly dependent on the origin load. Here, the invalidation-based and TTL-based approaches excel as they reduce the number of read requests without significantly impacting the write latency. CWC showed the poorest write performance due to the additional CRDT synchronization costs. However, the forum scenario demonstrated the biggest advantage of CWC, enabling Web proxies to directly process and respond to simple write requests, resulting in significantly lower latency than any other approach.

B. Cache Effectiveness

We previously discussed the benefits of caching REST APIs, but we did not evaluate the quality of the cached data. To address this, we look at additional caching metrics in Table I. It is important to note that the different application behaviors mean that the absolute measured numbers should only be compared within the same scenario.

The table reveals that TTL-based caching offered one of the best performance results with its 99% cache-hit ratio. However, the 5-minute inconsistency window and 98% staleness ratio make this option impractical for fast-changing APIs requiring up-to-date information. Advanced CWC offered the best performance after TTL-based caching, with a 10% lower throughput and 15% higher read latency. Despite this, it still resulted in a 26% higher throughput and 15% lower read latency than invalidation-based caching. However, CWC had around twice the write latency of TTL and invalidation-based caching in the flight scenario due to CRDT synchronization overhead. In the forum scenario, mutable CWC outperformed all other caching strategies because it was the only option that accelerated write requests. Compared to the second-best strategy, TTL-based caching, mutable CWC could reduce the mean write latency by 38%. However, the eventually consistent nature of the mutable CRDTs led to a higher staleness ratio (26%) compared to invalidation-based caching (17%) and no caching (1%). Interestingly, even without caching some responses were stale due to content changed during transmission.

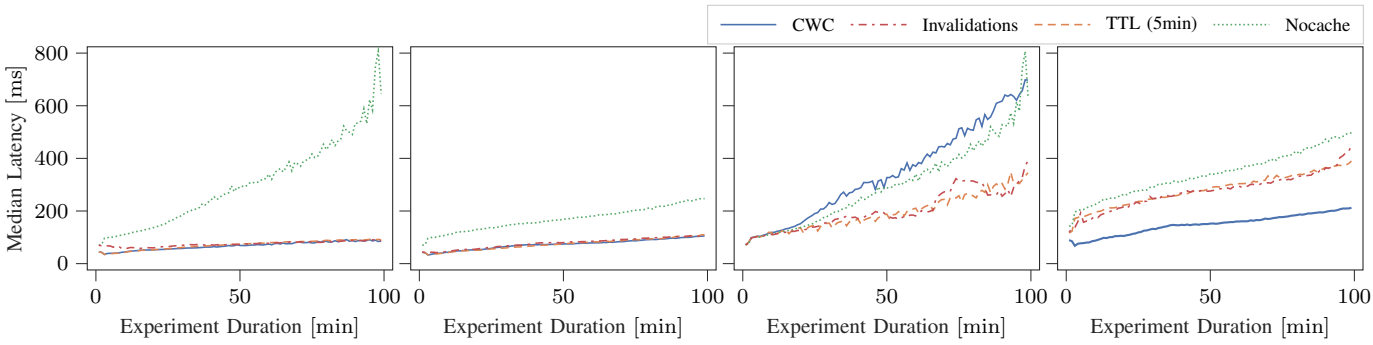
VII. DISCUSSION

This work proposes adding new data-aware functionality to existing Web proxies and we want to discuss some aspects.

a) Intended Scenarios: We do not think all REST API endpoints should adopt CWC because it adds complexity to the origin code and its maintainability. However, it can greatly benefit performance-critical endpoints. The basic and advanced level of CWC can be a valuable tool for developers to enhance the performance and scalability of read-only API endpoints where the chance of stale content should be low. If stale content is not a problem, TTL-based caching could be easier to realize. The applicability of mutable CWC is more limited as Web proxies cannot be aware of the application-specific semantic of concrete REST APIs. Hence, Web proxies can only perform basic mutations on the data, such as the CRUD operations (*e.g.*, inserting an entry into a list or updating an attribute). If these mutations are sufficient, mutable CWC can be a powerful tool to increase API performance.

b) CDNs in the Role of Information Brokers: We think that the enhancement of the current state of REST API caching is reliant on Web proxies becoming more powerful. At their core, REST APIs provide views on a data source, but the actual requests and responses are secondary. Proxies could significantly improve functionality by proxying the actual data (objects and relations) and not only requests and responses. This concept aligns with Information-Centric Networking (ICN) [24] principles. However, instead of proposing a new Internet Protocol we argue that already widely used standards like JSON:API and technologies like CRDTs can enable some of the advantages proposed by ICN. Application developers actively choose and configure a CDN; hence, they are in control of the functionality and can orchestrate the CDN to become brokers of their application data beyond being simple request-forwarders.

c) Transparent CWC: CRDTs were primarily designed for clients, *e.g.*, to enable collaborative editing [7]. Web proxies could provide clients with direct access to the endpoint CRDTs, but this adds new challenges such as security concerns. Web application clients are typically untrusted, and ensuring the integrity of changes made to CRDTs can be complex. While it might be possible to define permissions on CRDTs, this approach increases system complexity and has



(a) Read Requests - Flights (b) Read Requests - Forums (c) Write Request - Flights (d) Write Request - Forums

Figure 6. Median Latency per Minute. Every 1 min an additional client was started, increasing the system load.

debugging challenges. To minimize changes to existing systems and ensure the focus of the paper on caching challenges, we have kept the CRDT mechanism transparent to the client.

VIII. Conclusion

This work introduces CWC as a new method for distributing REST API data to Web proxies. We outline the requirements for achieving this on three levels: basic, advanced, and mutable. We evaluated the approach with a simple CDN deployment and two example applications (a flight booking service and a forum) in a Mininet simulation. Compared with no caching and state-of-the-art CDN caching strategies, we found that while TTL-based approaches can achieve better performance, they are not suited for the dynamic nature of REST APIs due to large inconsistency windows. Invalidation-based strategies where the origin actively purges outdated content from proxies can enable REST API caching. However, our approach outperformed this strategy as new content is automatically pushed to relevant proxies. While CWC demonstrated low read latency, it can come at the cost of higher write latency under high system load due to the overhead of the CRDT synchronization. However, mutable CWC outperformed every other strategy due to the acceleration of write requests.

Our work demonstrates how Web application performance can be enhanced by expanding the functionality of Web proxies and transitioning from simple HTTP request handling to more data-aware approaches. We argue that the existing semantics of JSON CRDTs and standards like JSON:API are sufficient to enable advanced data-aware proxies—the foundation of CWC for REST APIs.

We showed that CWC significantly benefits Web API performance in scenarios where it is vital to provide up-to-date information. In the future, CWC can help create specialized CDNs that focuses on the dynamic nature of Web APIs.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox, "Caching Proxies: Limitations and Potentials," in *Int. Web Conference (WWW)*, 1995.

[2] A. Bestavros, "Demand-based Document Dissemination for the World-Wide Web," Boston University, Tech. Rep., 1995.

[3] P. Cao and C. Liu, "Maintaining strong cache consistency in the World Wide Web," in *IEEE Trans. Comput.*, 1998.

[4] A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," in *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, USENIX Association, 1997.

[5] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Uni. of California, 2000.

[6] Y. Katz *et al.* "JSON:API," Accessed: Jun. 24, 2024. [Online]. Available: https://jsonapi.org/format/1.1/.

[7] M. Kleppmann and A. R. Beresford, "A Conflict-Free Replicated JSON Datatype," in *IEEE Trans. Parallel Distrib. Syst.*, 2017.

[8] M. Basalla, J. Schneider, M. Luksik, R. Jaakonmäki, and J. Vom Brocke, "On Latency of E-Commerce Platforms," *Journal of Organizational Computing and Electronic Commerce*, 2021.

[9] M. Sosnowski, R. von Seck, F. Wiedner, and G. Carle. "CRDT Web Caching: Additional Material. "[Online]. Available: https://tumi8.github.io/crdt-web-caching/.

[10] M. Fowler, *Patterns of Enterprise Application Architecture*. Pearson Education, 2012.

[11] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP Caching*, RFC 9111, 2022.

[12] F. Gessert, "Low latency for cloud data management," Ph.D. dissertation, University of Hamburg, 2018.

[13] B. Spang. "Building a Fast and Reliable Purging System," Accessed: Jun. 17, 2024. [Online]. Available: https://www.fastly.com/blog/building-fast-and-reliable-purging-system.

[14] Automerge contributors. "Automerge CRDT," Accessed: Jun. 17, 2024. [Online]. Available: https://automerge.org/.

[15] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm, "Enhancing the Web's Infrastructure: From Caching to Replication," *IEEE Internet Comput.*, 1997.

[16] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari, "Scalable consistency maintenance in content distribution networks using cooperative leases," *IEEE Trans. Knowl. Data Eng.*, 2003.

[17] B. Abolhassani, J. Tadrous, A. Eryilmaz, and S. Yüksel, "Optimal Push and Pull-Based Edge Caching For Dynamic Content," *IEEE/ACM Trans. Netw.*, 2024.

[18] W. Wingerath *et al.*, "Speed Kit: A Polyglot & GDPR-Compliant Approach For Caching Personalized Content," in *Proc. Int. Conference on Data Engineering (ICDE)*, 2020.

[19] M. Wan, A. Jönsson, and C. Wang, "Web user clustering and Web prefetching using Random Indexing with weight functions," *Knowledge and Information Systems*, 2011.

[20] Mininet Project Contributors. "Mininet," Accessed: Jun. 17, 2024. [Online]. Available: https://mininet.org/.

[21] G. Martinez, J. A. Hernandez, P. Reviriego, and P. Reinheimer, "Round Trip Time (RTT) Delay in the Internet: Analysis and Trends," *IEEE Network*, 2023.

[22] O. Victor Babasanmi and J. Chavula, "Measuring Cloud Latency in Africa," in *International Conference on Cloud Networking*, 2022.

[23] D. Bermbach, "Benchmarking Eventually Consistent Distributed Storage Systems," Ph.D. dissertation, KIT Scientific Publishing, 2014.

[24] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A Survey of Information-Centric Networking," *IEEE Commun. Mag.*, 2012.