Adaptive and Application-agnostic Caching in Service Meshes for Resilient Cloud Applications

Larsson, Lars; Tärneberg, William; Klein, Cristian; Kihl, Maria; Elmroth, Erik

Published in:
Proceedings of the 2021 IEEE Conference on Network Softwarization (NetSoft)

DOI:
10.1109/NetSoft51509.2021.9492576

2021

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Larsson, L., Tärneberg, W., Klein, C., Kihl, M., & Elmroth, E. (2021). Adaptive and Application-agnostic Caching in Service Meshes for Resilient Cloud Applications. In K. Shiomoto, Y-T. Kim, C. E. Rothenberg, B. Martini, E. Oki, B-Y. Choi, N. Kamiyama, & S. Secci (Eds.), *Proceedings of the 2021 IEEE Conference on Network Softwarization (NetSoft): Accelerating Network Softwarization in the Cognitive Age* (pp. 176-180). [9492576] IEEE - Institute of Electrical and Electronics Engineers Inc.. https://doi.org/10.1109/NetSoft51509.2021.9492576

Total number of authors:
5

Adaptive and Application-agnostic Caching in Service Meshes for Resilient Cloud Applications

Lars Larsson*, William Tärneberg†, Cristian Klein*, Maria Kihl†, and Erik Elmroth*
* Department of Computing Science, Umeå University, Sweden
† Department of Electrical and Information Technology, Lund University, Sweden

***Abstract*—Service meshes factor out code dealing with inter-micro-service communication.The overall resilience of a cloud application is improved if constituent micro-services return stale data, instead of no data at all. This paper proposes and implements application agnostic caching for micro services. While caching is widely employed for serving web service traffic, its usage in inter-micro-service communication is lacking. Micro-services responses are highly dynamic, which requires carefully choosing adaptive time-to-life caching algorithms. Our approach is application agnostic, is cloud native, and supports gRPC. We evaluate our approach and implementation using the micro-service benchmark by Google Cloud called Hipster Shop. Our approach results in caching of about 80% of requests. Results show the feasibility and efficiency of our approach, which encourages implementing caching in service meshes. Additionally, we make the code, experiments, and data publicly available.**

***Index Terms*—Service-mesh, Containerized network functions, Microservices**

I. Introduction

Micro-services have emerged as the dominant architectural design pattern for engineering scalable and resilient cloud applications. Said pattern encourages separation of concerns and data ownership between micro-services [1], thus, leading to frequent inter-service requests for data retrieval. In fact, a single public API request may cause orders of magnitude more inter-service requests.

Service meshes [2] have emerged to factor out commonalities in upstream and downstream communication between micro-services, such as load-balancing, retrying and graceful timeouts. This paper focuses on the **actuator** and proceeds to evaluate caching in service-meshes to improve resilience in cloud applications. While caching responses is not by itself novel, we are, to our knowledge, the first to evaluate using caching in service meshes, specifically gRPC. Indeed, caching responses is a well-known method for making web content delivery more responsive by reducing service and network load [3], [4]. However, with the exception of database caching, caching *in general* is not commonly used for inter-service communication. A key difference between web content caching and inter-service caching is that the latter has highly dynamic responses, that become stale after a few seconds, as opposed to days for web content. Hence, a key question is how the **time-to-live** (TTL) of a request affects data staleness and network traffic reduction of a realistic cloud application.

To keep the risk of stale data at acceptable levels, we re-purpose adaptive TTL estimation algorithms from the web content delivery field for this new purpose (for differences between the fields, please see Section II). To provide a quantitative evaluation of the system, we have selected two dynamic TTL estimation algorithms from the web content caching literature. The selection was driven by plausibility to work in the new context of inter-service communication, which has different properties than web content caching (Section IV).

To evaluate the two algorithms on a realistic cloud application, we implemented a gRPC-based caching infrastructure, mimicking a service mesh (Section III), and use it to empirically quantify the applicability of caching using dynamically estimated TTLs in Section VI. The experiments exposes the configured algorithms to a realistic micro-service setting (Section VI-A). The caching infrastructure is deployed with the Hipster Shop application developed by Google Cloud Platform and use their workload generator to subject the system to simulated e-commerce users. The contributions of this paper are as follows:

- We design and implementation caching in service meshes, which works even with gRPC; as expected from service meshes, the mechanism is application-agnostic and thus requires no source code changes;
- We demonstrate network traffic reduction with a real micro-service application.

The results (Section VI-A3) show that **about 80% of inter-service requests could be answered using cached data**, which also caused **an overall network traffic reduction by 40%**. Our work suggests that caching is a feasible and that it can be used as an efficient circuit breaker actuator, and encourages its implementation in service meshes. To facilitate reproducibility and reuse of results, we make all our source code and data sets openly available for benefit of the research community. Implementing additional algorithms is a straightforward process and requires very little code.

II. Background

Caching is extensively used in web content serving, and has been for a long time [3]. However, it is not commonly used in inter-service communication, and we believe there to be both technical and non-technical reasons for this. The

This work has been partially funded by the Wallenberg AI, Autonomous Systems and Software Program (WASP), the ELLIIT strategic research area on IT and mobile communications, Sweden's Innovation Agency (VINNOVA) under the 5G-PERFECTA Celtic Next project, the Swedish Foundation for Strategic Research under the SEC4FACTORY project.

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

technical ones are temporary hurdles to overcome through engineering: lack of support for caching certain HTTP verbs (gRPC uses POST for every operation, which is typically not considered cache-able), failure to communicate using the right transport protocol (HTTP/1.1 to upstream services rather than HTTP/2), etcetera. All these can be solved rather easily and be incorporated in software. While our work focuses on gRPC, which has no support for caching in its specification (in spite of nascent support in the Protobuf service descriptor for marking operations as idempotent [5]), it should be noted that there are no *technical reasons* that prevent typical REST-based services from using well-established HTTP/1.1-based caching infrastructure. It seems to be not commonly done in practice. For example, major vendors such as Microsoft does not mention it in their REST API guidelines [6].

The non-technical reasons are more interesting to us, as the major hurdle does not seem to be the technical challenges. A reason that cannot be ignored is that **it is non-trivial to *a priori* determine TTLs for responses**. Software developers cannot during development reasonably know for how long responses will be valid, unless the underlying data is known to be stable for some time (e.g. weather estimates that are updated hourly). But letting software inspect responses and thereafter estimate TTLs during run-time is definitely possible.

Determining *which* operations are possible to cache can also present a challenge. It is generally considered good API design to separate operations that can mutate state from the ones that cannot. REST enforces this via HTTP verb mapping [7]. Because gRPC lacks caching on the protocol level, there is no such enforcement. Still, it is an ingrained best practice design pattern and developers and operators are therefore generally aware of which operations can mutate state and can therefore inform software of it.

It is a generally accepted practice to use a fast in-memory key-value store such as Redis in front of databases for read queries to avoid needlessly straining the database service with possibly complex queries (e.g., ones requiring multi-table JOINs) [8], [9]. The application code is then adapted to always check the key-value store *before* issuing the possibly complicated database query, where the results may be cached. Thus, it is up to application developers to not only decide which operations to cache but also, possibly, for how long. The approach we take differs in that we (a) cache in-between services, not just in front of the canonical database server; (b) require no application awareness of caching — as, indeed, gRPC applications have no concept of caching; and (c) object cache time-to-live is continuously re-estimated. In this way, applications can get the benefits of caching across micro-service architectures where calls are performed in many steps before hitting a database, and application developers need not make their applications cache-aware.

Services that use gRPC for inter-service communication often expose a REST interface toward clients. Would it therefore not be sufficient to cache only the client-facing responses? We argue that it is **not sufficient** in a micro-service application, for two reasons. Firstly, modern services typically have analytics and other batch jobs that rely on direct inter-service requests, rather than on publicly facing aggregated APIs. Second, TTLs for aggregated results are bounded by the lowest TTL among the constituent sub-results. Our results with the Hipster Shop application show that, on average, a single client request branches out and requires aggregated data from around 13 inter-service requests (Section VI-A4). Should even a single of these have low TTL and the others a high TTL, it would invalidate the aggregated response and all requests would have to be wastefully re-issued if only client-facing caching was used. This has been previously explored with regard to personalized web sites in e.g. [10].

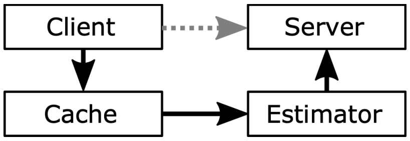

Figure 1: Caching infrastructure architecture overview showing the old traffic flow as a dotted gray line and the new traffic flow through Cache and Estimator components in black.

Inter-service communication differs from web content caching, we therefore regard the following properties as important differences:

- updates are potentially more frequent, and TTLs therefore shorter. Well-designed web applications consist of immutable and therefore infinitely cache-able static resources, presented via a dynamic HTML page, making the orders of magnitude smaller (in bytes) HTML page the only asset that needs a short TTL;
- large variance in object popularity. Unlike web content caching, where some objects are much more popular than others due to human preference (70% of objects at a CDN were requested only once over a multi-day period [11]), API requests are highly varied and popularity distribution need not be tied to human preference; and
- calculations and responses must be fast. Because client-facing requests cause multiple inter-service requests, caching must not add significant delays, lest the multiplicative effect be noticeable.

III. Caching Architecture

Service meshes instantiate for each micro-service two proxies: one handling upstream calls and one handling downstream calls. This allows the service mesh to intercept all inter-service calls in an application-agnostic way, and offer higher-level communication functionality, such as circuit breaking. We hereby propose a system architecture that can readily be deployed in a service mesh.

Our proposed caching actuator has two components: the *Estimator* and the *Cache*. Respectively, they are responsible for estimating for how long a response object is valid and for caching responses for (maximally) that amount of time.

Figure 1 shows the conceptual architecture and traffic flow in the system. Instead of direct connection between the Client and Server (dotted gray line), the newly added components are deployed and configured to intercept the traffic (black lines).

The Estimator and Cache components can be deployed in different configurations, i.e., in the downstream or upstream proxy of the service mesh, each favoring different aspects of a performance and cache coherency trade-off. These are discussed in Section III-C.

A. Cache component

Unlike HTTP/1.1, where caching is specified as part of the protocol [12], gRPC has no notion of caching (see also Section II). Accordingly, the Cache component in our proposed system must respond as the Client expects a Server to respond. This makes the Cache behave indistinguishably from a Server from the point of view of the Client, thereby allowing for seamless integration with existing gRPC applications.

It is valid to add metadata in headers for gRPC responses. We use the `Cache-Control` header to express the TTL in seconds, similar to how HTTP/1.1 defines it. If response TTL is given in the header of a response, the Cache component will cache the response for the given amount of time. If not, or the TTL is specified as 0, the response will not be cached.

B. Estimator component

The Estimator component estimates how long a response to a particular request can be considered valid and therefore cached. Since gRPC Servers do not typically convey how long responses are valid, the Estimator can use multiple different algorithms to estimate object cache validity (see Section IV).

When a request has been made to the Estimator, it will for a limited duration of time produce response TTL estimates for subsequent equivalent requests. The time limit is used for housekeeping purposes: once the time limit is surpassed, the Estimator will de-allocate the memory used to calculate estimates for the particular request.

Because the Estimator cannot know when a response to a request has changed, it has to continuously update its estimates. The Estimator will contact the upstream Server whenever it gets an incoming request. The reason for an incoming request must be that the Cache cannot answer a Client request from memory, which either means that the Cache has restarted or the response TTL has been surpassed. Regardless, the Estimator will contact the upstream Server and make a new TTL estimate.

C. Component co-deployment

Because the Cache and Estimator components are designed to seamlessly deploy into the network between Client and Server, a number of different deployment scenarios are possible. In this work, we focus solely on the case where Cache components are *co-deployed* with Clients, and Estimator components with Servers. We defer investigation into the consequences of the different deployment scenarios with regard to, e.g., cache consistency and traffic reduction to future work. In practical terms and in the context of this work, co-deployment means that a *sidecar* container is started in a Kubernetes Pod. By definition, this implies that localhost networking can be used between co-deployed components. This follows an established pattern of how, e.g., service meshes such as Istio.

IV. TTL ESTIMATION ALGORITHMS

Meeting the requirements stated in the previous section and cognizant of differences between web content serving and inter-service request handling, we have implemented to algorithms that take very little memory and require no large body of training data to function. For comparison reasons, we have also implemented a simple static TTL "estimation" as well. The three TTL estimation algorithms used in this papers are presented in Table I.

V. IMPLEMENTATION

Our design goals for the implementation are to be extensible, suitable for research via instrumentation/observability, and easy to integrate with existing service meshes. This implies an application-agnostic approach, such that existing gRPC-based services can benefit from it without source code modifications.

Extensibility is ensured via implementing the Estimator and Cache gRPC *interceptors*, i.e., as plugins that capture and possibly modify requests before they are passed along to the intended process.

Instrumentation/observability for research is implemented by letting interceptor output timestamped CSV rows with nanosecond resolution. All operations output the name of the invoked method, TTL, timestamp and whether a response had to be passed upstream to the Estimator or could be answered using cached data. Together, the data can be used to form a picture of overall system performance.

Application-agnosticism and the ability to use our caching infrastructure without source code modification is enabled by attaching the Cache and Estimator gRPC interceptors to purpose-built reverse proxies. The code for these is auto-generated from the Protobuf service descriptor using our modified version of the gRPC code stub generator.

VI. EVALUATION

The objective of this section is to establish a set of experiments that will validate the proposed caching infrastructure and its implementation. Further, because caching always introduces a risk of stale data to achieve a reduction in network traffic, the inherent trade-offs in our proposed system must be evaluated and addressed. In particular, we must establish which trade-offs are provided by which algorithm configurations and whether the dynamic caching and supporting infrastructure approach work for real micro-service applications.

To validate the caching infrastructure in a real setting, we perform experiments with a real micro-service application. We have chosen the "Hipster Store" by Google Cloud Platform. The focus of this experiment is two-fold: (a) to verify that our caching infrastructure works for a micro-service application without any source code modification; and (b) to see what network traffic reductions can be made using caching and conservatively configured dynamic TTL estimation algorithms.

Data staleness, while often favorable compared to non-responsive services, is generally to be avoided. However, what level of staleness is acceptable is application-dependent: certain values are never allowed to be stale (e.g. a customer's

Name	Formulation: $\mathrm{TTL}_x =$	Ref.	Description
Static TTL	β	n/a	β is the number of seconds (integer) to always statically respond with as response TTL for each incoming request. Setting the β parameter to zero implies that no caching should be made
Adaptive TTL	$(t - M_x) \times \alpha$	[13]	α is real number that, while technically semantic-free [13], practically signifies a linear "acceptance" of stale data by the operator. Higher values of α mean longer estimated TTL, and thus, higher risk of stale data. M_x is the time the object was last modified.
Update-risk based TTL	$-\frac{\mathrm{BUD}_x(K)}{K} \log(1 - \rho_x)$	[13]	This algorithm takes as a parameter an operator-specified *acceptable update risk*, $\rho_x \in [0, 1)$, for a given object x. $\mathrm{BUD}_x(K)$ is the "backward K-update distance", the time of the K^{th} most recent response object update of x.

Table I: The three TTL estimation algorithms used in the for evaluating the proposed caching architecture.

order history), whereas others are less critical (e.g. a product recommendation). We do not quantify data staleness as part of the our suite of experiments, but rather, conduct the experiments using only conservatively configured TTL estimation algorithms to keep staleness as generally low as possible.

For general applicability, we do not explicitly focus on latency or response times. Latency and response times are nonlinear functions of the amount of work that a server has to do [14] and depend on a multitude of factors, such as application code, its deployment, and the underlying hardware resources, the confluence of which causes unexpected behavior in both the application and the control plane [15]. Thus, a more objective and general measurement on algorithm efficiency and performance is to consider the number of requests that are transmitted across the network and, when a specific application is used, the number of bytes such transmissions consist of. Unless, of course, caching *itself* would add considerable processing time — however, our choice of algorithms and results strongly indicate that this is not the case here.

A. Quantifying network load reduction in a real application

Although it would be desirable to evaluate data staleness in this experiment too, this is non-trivial to achieve. For example, if the client pays for an item to the shopping cart, this not only affects the Cart micro-service, but also a number of others, e.g., the ShippingService. Hence, even if we modified the client to remember the last value set for each API call, side-effects across micro-services prevent the client from accurately predicting the freshest value for a different API call.

Hipster Shop, chosen for our evaluation, is a polyglot application with 11 micro-services that communicate over gRPC. Note that **no source code has been modified** in Hipster Shop, we only modified the Kubernetes deployment manifest. These modifications are simple and can be automated.

To each service in Hipster Shop, we added an Estimator. Because not all requests can be cached, we blacklisted certain requests (see also Section VI-A2). The Estimator components were all given the same configuration, depending on which algorithm was under test.

Cache components were added to the three services in Hipster Shop that perform inter-service calls over gRPC: Frontend, CheckoutService, and RecommendationService. Adding a Cache component to other services would not affect them in any way, apart from wasting resources on a component that would be dormant. Note that we **do not cache non-gRPC traffic**, i.e. the HTTP responses to the Load Generator and the Redis communication initiated by the CartService.

1) Load generation: Hipster Shop ships with its own load generator. The load generator operates in closed-loop manner [16] and waits a random amount of time (uniform distribution) before issuing the next request. The set of possible requests is pre-defined, and weights are attached to the requests, which affects the probability that a particular request is randomly chosen more or less often than the others.

2) Cacheable subset of operations: Not all operations in Hipster Shop can be cached without introducing significant application-level errors. Caching, e.g., a call to `AddItem(user_id, item)` such that a repeated call would be ignored by the CartService would be highly detrimental to the application.To mitigate this, we used cache black-listing to disallow caching of state-modifying calls.

3) Results: network traffic reduction for a real micro-service application: The results of the experiments are presented below, followed by an analysis in Section VI-A4. Based on the results obtained by the previous sets of experiments, we deployed Hipster Shop in a Kubernetes cluster (minikube instance) with our caching infrastructure. The two most conservative dynamic TTL estimation algorithm configurations were deployed ("dynamic-adaptive-0.1" and "dynamic-updaterisk-0.1") as well as the no-caching baseline ("static-0").

Figure 2 shows the amount of network traffic for the three algorithm configurations, averaged in 15-second increments during the three experiment repetitions. The amount of network traffic is visibly clearly reduced when caching is used, in comparison to when it is not. Both caching algorithms achieve a traffic reduction of about 40%, with the very slight advantage going to the Adaptive TTL algorithm.

What caused the traffic reduction is the use of cached data. The total number of requests in the application are very similar across experiments (differences related to randomness in the load generator, see Section VI-A1) and both dynamic algorithms manage to cache about 80% of responses. As stated in Section VI-A2, not all requests can be cached. Service responses are of course also not equal in size, which explains why an 80% reduction in requests could translate into a 40% reduction in network traffic.

4) Traffic analysis: Analysis of the Hipster Shop experiment log files show that 12% of requests were not initiated by the Frontend service. Two other services (RecommendationService and CheckoutService) also make requests to carry out their work. This implies that inter-service requests that can be answered from cache between these services and the ones they request data from benefit from not only caching at the publicly facing Frontend.

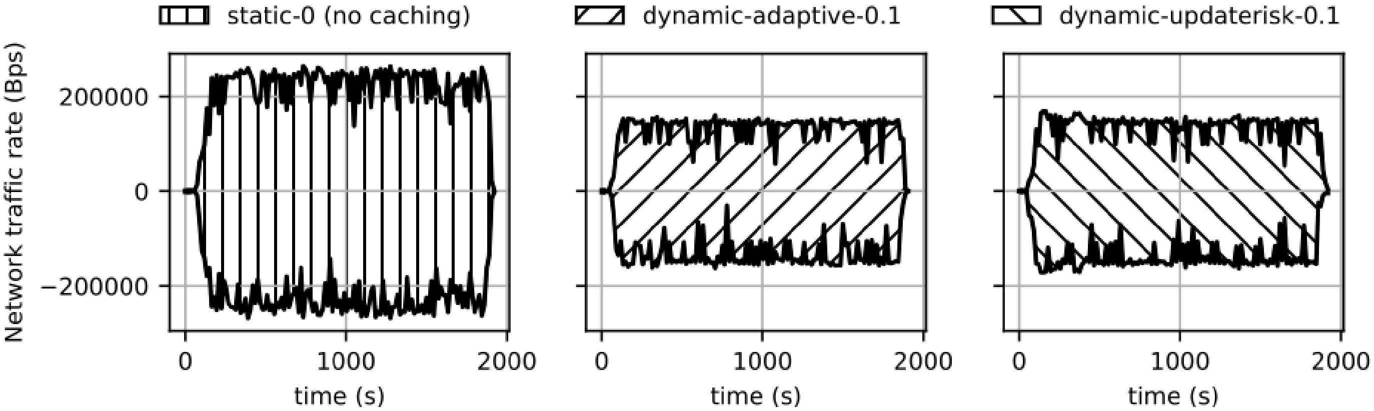

Figure 2: Network traffic over time (bytes/second) in Hipster Shop for when no caching is used (**static-0**), and for the two most conservative dynamic estimation algorithm (**dynamic-adaptive-0.1** and **dynamic-updaterisk-0.1**). Outgoing traffic is shown as positive, incoming as negative. Thinner overall shape therefore implies less traffic.

Although we purposefully did not seek out to include response time analysis in these experiments (for good reason: the minikube Kubernetes cluster is far from a production-ready or realistic execution environment), it is worth noting that response time for most operations was cut in half with caching enabled compared to when it was not (not shown for briefness and to not place undue focus on it in this evaluation).

VII. Conclusion

The micro-services paradigm dictates a separation of concerns and strict data ownership, which implies that services must interact frequently with each other across the network. Such communication is increasingly dealt with by service meshes, which uniformly implement features such as load-balancing and retrying.

In this work, we proposed caching for service meshes. We estimate cache TTLs for responses dynamically using adaptive algorithms from the literature on serving web content. We have evaluated our approach and found that in spite of frequent updates, conservative configuration of dynamically estimated TTL estimation algorithms could keep data staleness at 0–3% while reducing load by up to 30%. When used in a realistic off-the-shelf e-commerce micro-service application, 80% of the requests were served cached responses with 40% fewer bytes transferred.

Open Source and Open Data Notice

The source code and data sets used in this work are available in the following locations:

- https://github.com/llarsson/grpc-caching-interceptors, hosts the gRPC interceptors that implement the Caching and TTL Estimation.
- https://github.com/llarsson/protobuf, contains a modified Protobuf compiler that provides a reverse proxy server.
- https://github.com/llarsson/hipster-shop, is the Hipster Shop application and our scripts and Kubernetes manifests for running experiments.
- https://github.com/llarsson/hipster-shop-experiments, contains the data set from the second suite of experiments.

References

[1] J. Lewis and M. Fowler, “Microservices,” 3 2014, library Catalog: martinfowler.com. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in *Int. Conf. on Service-Oriented System Engineering (SOSE)*. IEEE, 2019.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, “A hierarchical internet object cache.” in *Annual Technical Conf.* USENIX, 1996.

[4] A. Iyengar and J. Challenger, “Improving web server performance by caching dynamic data,” in *Symposium on Internet Technologies and Systems*. USENIX, 1997.

[5] gRPC Authors, “gRPC over HTTP/2 (version 1.28.x),” 12 2019. [Online]. Available: https://github.com/grpc/grpc/blob/v1.28.x/doc/PROTOCOL-HTTP2.md

[6] Microsoft, “Microsoft REST API guidelines,” 2020, visited March 26, 2020. [Online]. Available: https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md

[7] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[8] B. Familiar, “Iot and microservices,” in *Microservices, IoT, and Azure*. Springer, 2015.

[9] K. Brown and B. Woolf, “Implementation patterns for microservices architectures,” in *Conf. on Pattern Languages of Programs*. The Hillside Group.

[10] W. Shi, R. Wright, E. Collins, and V. Karamcheti, “Workload characterization of a personalized web site and its implications for dynamic content caching,” in *Int. Workshop on Web Caching and Content Distribution (WCW’02)*, 2002.

[11] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman, “Adaptive TTL-Based Caching for Content Delivery,” in *Int. Conf. on Measurement and Modeling of Computer Systems*. ACM, 2018.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “ Hypertext Transfer Protocol – HTTP/1.1 ,” Internet Requests for Comments, The Internet Society, RFC 2616, 6 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2616.txt

[13] Jeong-Joon Lee, Kyu-Young Whang, Byung Suk Lee, and Ji-Woong Chang, “An update-risk based approach to ttl estimation in web caching,” in *Int. Conf. on Web Information Systems Engineering, (WISE)*. IEEE, 2002.

[14] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, “Web server performance modeling using an M/G/1/K*PS queue,” in *Int. Conf. on Telecommunications, (ICT)*. IEEE, 2003.

[15] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, “Impact of etcd deployment on kubernetes, istio, and application performance,” *Wiley, Software: Practice and Experience*, 2020.

[16] B. Schroeder, A. Wierman, and M. Harchol-Balter, “Open versus closed: A cautionary tale.” USENIX, 2006.