UC Santa Cruz

UC Santa Cruz Previously Published Works

Title

Flexible Evaluation Caching Using FATE

Permalink

<https://escholarship.org/uc/item/0pp2s187>

Authors

Mathewson, James
Garcia-Luna-Aceves, J.J.

Publication Date

2019

Peer reviewed

Flexible Evaluation Caching Using FATE

James Mathewson,^{*,†} and J.J. Garcia-Luna-Aceves^{*,†}

^{*} Computer Science and Engineering Department, University of California, Santa Cruz, CA 95064

[†]Akamai Technologies Inc., Santa Clara, CA 95054

[†]Palo Alto Research Center, Palo Alto, CA 94304

Email: jmathews@akamai.com, jj@soe.ucsc.edu

Abstract—Many algorithms have been presented to enhance caching in the Internet, and LRU (Least Recently Used) is the de-facto standard used to evict data due to its ease of implementation and good performance. A limitation in creating a new caching algorithm is the cost associated with its development, testing, analysis, and efficiency in specific scenarios. The Functional Algebraic aTomic Evaluators (FATE) is introduced to address this problem. FATE is an information processing engine that evaluates information (such as the value of cached content) and acts upon said results. FATE is designed to be highly configurable and flexible, and to enable rapid development, accuracy in duplicate functionality on different platforms, and ease of use. It uses XML configuration files to setup the evaluation of the content. Each algorithm in FATE can be used as-is or optimized.

I. INTRODUCTION

Developing a new caching protocol for the Internet can be an arduous process. During or after its design, simulation analysis is typically needed to understand its performance. This step is complex due to the difficulty in implementing code specific to each architecture and simulator. One example of this is the simulator code for ns-3, which does not work with ccnSim [3]), which uses omnet++ [15] as a base environment. These problems do not even consider different networking or environmental constraints, nor the modifications necessary to modify a protocol to cover corner cases.

For academia, simulations are compared over various protocols, given a set of constraints. This is complicated by the availability of other protocols, for comparison, on the host platform; which may require reimplementing on another platform, or a reimplementing of the competing algorithm, which may be prone to interpretation or other errors.

Reproducibility [7], [14] is a major hurdle in the analysis of caching protocols and algorithms. A network-caching protocol should give similar results when running on the same simulator; however, many times, the protocol code is not available, requiring rewriting an implementation based upon the given protocol in a paper, when used for comparison. In addition, the same protocol, if available, may only be available on a single simulation platform, requiring either a rewrite to a new platform or implementing the new protocol to a potentially undesirable platform. All of these problems affect reproducibility.

To help remedy these deficiencies, we introduce Functional Algebraic aTomic Evaluators or FATE for short. FATE is designed to facilitate rapid development of networking algorithms on various platforms. Though FATE, technically, is an Information Centric processing and decision engine (making

decisions based upon multiple evaluators), for the purpose of this document, it will only be discussed as it is used for caching content in a network. FATE is not a network simulator (like ns3) but is an information processing unit utilized for security, caching, and forwarding, as applied to networks. FATE helps solve the reproducibility problem by evaluating via algebraic equations and performing an action based on said evaluation. FATE uses the c++11 standard template library, to minimize requirements for external libraries. In FATE, changes are simply using the correct algebraic atomic functions, defined in an XML file, in the correct logical manner. Only new evaluators may need to be created, but once created, can be shared and reused, reducing redevelopment of code for evolutionary changes.

II. RELATED WORK

When a cache becomes full, it is necessary to evict content, to make room for new content. While FATE uses a ranking function, and based upon the evaluation, it will evict stale content. The most popular eviction strategies are to evict the least recently used content first (LRU) or to evict the least frequently used content first (LFU).

Many alternative caching algorithms have been published, each having some performance benefit under various network constraints. The problem is to identify which properties of an algorithm perform best and under which conditions.

Probabilistic Caching [13] attempts to solve the problems associated with the inefficiency of multiple on-path caching nodes. ProbeCache resolves this problem by using a Cache Weight system to determine, probabilistically, if content should be cached at the node.

Laoutaris [8] approached the problem from a different angle, and showed moving cached content closer, on an on-path network, per node, improved results over probabilistic caching. The two algorithms, LCD (Leave Copy Down) and MCD (Move copy down), where each 'hit' moves the content one (cached) hop closer to the content.

Another algorithm, proposed by Lee [9], suggested LRFU (Least Recently Frequently Used) as a cache eviction algorithm. LRFU, mathematically, can be modeled as LRU, LFU, or a value in between, with a complexity similar to LFU.

NS-3 [4] is an event driven simulator, using C++ and python code to schedule events. It is one of the largest supported simulators, with thousands of papers published on its platform. The strength of NS3 is that it is event driven and allows multiple processes to be run on the correct platform, thus

giving results in significantly faster time frame. FATE does not replace a network simulator, but offers an alternative Node (switch) implementation. NDN [1], [10] is built over NS3, and is used as the base platform for our presented results, due to its maturity and support for caching algorithms (natively, NS3 does not support caching). We note that FATE does not require NS3/NDN as a platform, but was chosen as a mature open source networking simulator (NS3) with caching support (NDNSim).

III. FATE CACHE IMPLEMENTATION

FATE introduces several concepts to assist in rapid development and testing of network protocols and algorithms. First, the code, is written in C++11, is agnostic of the platform (ns-3, qualnet, linux, etc), and can be used in an information-centric networking (ICN) architecture or the IP Internet architecture. To resolve system dependencies, the code is written to use various resources, such as a timers, which are wrapped around the model. For example, on a linux platform, the linux timer is called, on an ns-3 simulator platform, its native timer is called, to be independent of any external resource.

FATE is organized in a top-down format. Specific-purpose modules can be defined for any purpose, with the intent to have the specific module do its own job for easier maintenance, verification, and testing. Some modules are forwarding, caching, discovery, and security. In this paper, a modified NDN ContentStore was used to work within the existing architecture. Each utility is an atomic algebraic function that evaluates the content, or returned value, and returns a normalized scalar ($[0, 1]$). This allows a very flexible and powerful method to evaluate information. The module takes a valuation of the content, and performs an action on it (e.g., caching stores or evicts content, forwarding decides which packet for which egress port, security evaluates the packet for trust-worthiness). In this paper, we concentrate on the caching aspects of FATE.

Most networking nodes evaluate content (or packets) based upon attributes of that content (e.g., when it was received for LRU caching, freshness lifetime for html content, TTL field in IP), and make an algorithmic decision based upon a specified criteria. In FATE, each component (e.g., LRU, TTL, FRESH) are atomic functional evaluators. Based upon the algebraic formulation provided, a decision is made on the final evaluation. Thus, FATE evicts content or not based upon the results of the evaluation. FATE is meant to allow rapid prototyping of algorithmic features, and be portable over any platform to aid in research. FATE can be used, as-is, in various platforms for security, forwarding, or caching. However, we note that FATE will be slower than a customized algorithm (due to the evaluations) and, most likely, use more memory (each evaluator carries internal state, with no external dependencies). As an example, LRU needs no internal state (unlike LFU), by using its ordering to dictate which content was used least, resulting in easier evictions.

FATE uses a modified BSD license. The license, makes the code free to use, with the exception of giving credit when FATE or a part of it is used.

A. Functional Algebraic aTomic Evaluators

The concept of FATE is to evaluate information (typically via named packets of information), and perform an action, based upon said result. To evaluate a result, atomic algebraic functions are used. Each function can be an aggregate (such as minimum or addition), or an atomic evaluator. An atomic evaluator can evaluate based upon a content (such as a meta-data attribute like hop count, or type of service), context (such as the purpose of the packet, e.g. interest or data), or by name (use a function, such as Least Recently Used, evaluates upon). Each function (atomic or combinational) returns a normalized scalar $[0, 1]$ that allows each function to return a normalized value no matter how dissimilar in evaluation, which can be compared or evaluated with each other. Since all functions return a normalized scalar, actions are based upon matching an expected range. Any content may qualify for caching, but only high value content should be cached, as determined by FATE evaluators.

1) *Aggregation Functions*: FATE supports several aggregation methods that take one or more inputs, and return appropriate results. The following is a partial listing of available aggregation functions.

MIN : MINIMUM(a,b,...,z) return the minimum value of its inputs.

MAX : MAXIMUM(a,b,...,z) return the maximum value of its inputs.

ADD : ADDITION(a,b,...,z) return the sum of all its input. The sum may be greater than 1.0, and may require scaling.

MULT : MULTIPLICATION(a,b,...,z) return the product of its inputs.

STEP : STEP(function,range) if 'function' has a value with the specified range, return a value of 1.0, otherwise return a value of 0.0. It is also known as an impulse or (single) step function.

2) *Atomic Methods*: FATE supports several atomic methods. Some methods have an option of how to rank information based upon configuration settings (such as LRU, which may be evaluated temporally or spatially). Each atomic method may be stateful, but the state is exclusive to each instance of the method. Atomic methods evaluate a specific attribute, functionality, algorithm response, or statistical method, with a specific purpose, to provide an evaluation based upon its functionality. As an example, certain algorithms are based upon several or multiple parameters; whereas FATE is based upon the principle of having many singular functions do the evaluation, then obtain a weighted result based upon the appropriate aggregate function). The following is a subset of atomic algebraic methods currently available.

LRU : LRU, or Least Recently Used, Ranks the most recent information, with the highest value (1.0), and progressively lower ranked information has a lower value. LRU has two ways to rank itself, one is temporal and the other is spatial (figure 1). Spatial ranking is evaluated by when the Information was received (e.g. 5 pieces of Information will be ranked at: 1.0, 0.8, 0.6, 0.4 and 0.2). The evaluated difference between evaluations is constant. The other method to measure LRU is

temporally, which evaluates based upon *when* the information is received. As depicted in the graph, after 0,1,3,8, and 10 seconds, the information is evaluated at; 1.0, 0.9, 0.7, 0.2 and 0. Depending on the method of evaluation, the Information may be (or never be) evaluated at zero.

Fig. 1. Temporal vs Spatial LRU values

LFU : LFU, or Least Frequently Used, ranks the highest occurrence of information with the highest value (1.0), and lower occurrence information, progressively less. LFU, like LRU, can be evaluated in different methods. LFU can be evaluated as spatial or weighted ranking (see figure 2). As shown in the graph, and like LRU, spatial ranking will have a constant differential value between Information evaluations. The other method of evaluating is Weighted (based upon actual number of occurrences). In the graph, based upon the number of occurrences (or hits in a cache system), of 1,4,5,6, and 10 occurrences, weighted LRU will evaluate the Information as 0, 0.4, 0.6, and 1.0, respectively.

Fig. 2. Weighted vs Spatial LFU values

CONSTANT : A constant value, typically used with multiplication, e.g. $0.5 * LRU$.

HASH : $HASH(rawdata, modulus)$ uses a modulus type function, based upon a hash value of either an attribute, data, packet, or file. If the modulus of the hash matches a configured value, it returns a '1.0', otherwise it returns a '0.0'; $5 \% 3 == 2$.

FRESH : $FRESH(packet.attribute)$ uses an attribute of the packet (such as html Cache-Control metadata), to define if

the file can be cached, and for how long. Evaluation of this attribute results in a '1.0' for fresh, and '0.0' for stale.

HOPCNT : HOPCNT simple counts (up or down) from a specified node (e.g. client or server), to identify shorter and longer routes. Useful for caching content which has a significant penalty for cache misses.

Since all rankings occur within the $[0,1]$ range, some values are calculated based upon a secondary metric as mentioned above. The metric is defined by the function itself. Both LRU and LFU algorithms have a spatial implementation, but differ, based upon how the algorithm is defined, to use either a temporal or weighted implementable metric.

Some of the choice on which implementation of an algorithm depends on possible memory or computation intensity for said algorithm. Spatial ordering may be memory intensive as the ranking of each content, along with the original time stamp is kept (and may be recomputed). Temporal and Weighted ordering, are based upon the highest and lowest measurement value, and only need to be computed when either of these values are removed (adding a higher measurement does not change the internal marker. But the removal of said measurement will require a search of all content, to find the updated low/high measurements).

Fig. 3. Utility Caching Tree representation of $MIN(0.6*HASH+0.4*LRU, FRESH)$

B. Modules

The structure of FATE can be viewed as a tree, as illustrated in Figure 3, with atomic functions being the leaf, and algebraic aggregate functions being branches. The information packet is passed down and evaluated at each leaf. The results then move along the aggregated branches, towards the root, to give an evaluated result. To easily model this, we need two main (atomic) functions: the traditional caching algorithm to determine content value is determined by a weighted evaluation from LRU. A bias for node placement is determined by the HASH of the content, which if matched will give greater value for the packet to be stored in a matching node, as opposed to a non HASH matching node. Finally, FRESH allows non fresh

content to be evicted. It should be noted, that when the cache (or content store) exceeds a defined threshold, it will evaluate all existing content and purge all the lowest value content. Thus, if a content store of size 10, adds new content, the entire store is evaluated (including the new content). If the new content can not be cached (valuation of '0'), or any content expired (stale, not fresh), will be evicted. From this formula, it may useful for having some content cached (due to the hashing function), using an LRU to decide the value of the packets. Notice new content will have a lower value (0.4 at best), compared to low value, correctly placed (via HASH) packets (0.6 minimum). FRESH removes all expired/stale content.

IV. RESULTS

FATE caching was evaluated using two simple network scenarios. Figure 4 illustrates the first scenario of a simple network with a single caching node between a client and a server. Figure 5 illustrates the second scenario, which consists of a similar setup, except that four on-path caching nodes exist between the client and server. Both scenarios were run with five different seeds (the averages are presented), using a zipf-mandelbrot distribution ($s=0.7$), with $N=10000$ (maximum number of uniquely requested content). Each scenario was run for 1000 seconds, with a request rate of 100 requests/sec (for 100k requests total). Each caching node can contain 10 elements.

Fig. 4. Simple single caching node

Fig. 5. Four Caching Node Network

There is no consensus on what the correct Internet traffic distribution is, with some support for a Mandelbrot distribution [6], while others support a Zipf distribution [2]. Both distribution models were tested using the same random seeds on the single cache scenario by using different 'q' settings in the Mandelbrot distribution. Table I uses the default adopted in NDNSim with a 'q' setting of '0.7' for the Mandelbrot distribution. Table II shows the results of the Mandelbrot 'q'

setting set to '0', which closely resembles a traditional Zipf distribution.

The LRU and LFU algorithms implemented in NDNSim are used as a benchmark against algorithm FATE1 (LRU implemented in FATE). FATE1 is contrasted with LRU implemented in NDN to demonstrate that the algorithmic and evaluated implementations of LRU return the same results. FATE2 is a weighted combination of LRU and LFU given by the algebraic formula $(0.2+LRU*0.8)*LFU$. The FATE2 algorithm is implemented to give the benefit of LFU at higher hit rates while allowing the temporal property of LRU (as popular content changes, the older and more popular content is purged, unlike with LFU). This combination gives nearly a 28% and 12% average increase in hit rate, respectively. Other combinations of LRU and LFU were tested (e.g., adding or using different weights) but no other combinations were as successful.

TABLE I
SINGLE CACHE HIT RESULTS BY ALGORITHM, Q=0.7

ALGORITHM	MIN	MAX	AVG
LRU	0.84%	0.90%	0.88%
LFU	3.75%	5.59%	4.34%
FATE1	0.84%	0.90%	0.88%
FATE2	5.01%	6.47%	5.54%

TABLE II
SINGLE CACHE HIT RESULTS BY ALGORITHM, Q=0.0

ALGORITHM	MIN	MAX	AVG
LRU	1.10%	1.18%	1.14%
LFU	4.15%	6.71%	5.90%
FATE1	1.10%	1.18%	1.14%
FATE2	6.17%	6.97%	6.61%

The results show that the atomic FATE LRU evaluator ranked values similarly to the traditional algorithmic LRU implementation in NDN. LFU significantly outperforms LRU, but LFU suffers from a temporal problem. If content *was* very popular, but not currently popular, it remains in the cache. LRU does not suffer from this problem. In correlation to LRU and LFU, the FATE cache evaluator of $((0.2+LRU*0.8)*LFU)$ gives a slightly better hit cache ratio than a pure LFU, while retaining the temporal properties of LRU as content popularities change. If popular content is not used for a while, it is replaced as the LRU values declines over time.

According to results published by Dabirmoghaddam et al.[5], caching is most effective when it is placed closer to the requesting client, and on-path caching provides no benefits compared to a larger cache near the edge.

In Scenario 2, LFU does take advantage of the additional caching between client and server compared to LRU, which barely improves with additional on-path caching. However, the hit rate declines as the distance from the client increases. Noticing this trend, we have created and tested three custom algorithms, each giving a better caching efficiency, better content response, and an intermediate results of the two.

TABLE III
NETWORK WITH FOUR ON-PATH CACHING NODES

ALGO	C1 Hit Rate	C2 Hit Rate	C3 Hit Rate	C4 Hit Rate	Total Hitrate	Total Resp Delay(ms)
LRU	0.86%	0.16%	0.17%	0.15%	1.35%	491598
LFU	4.58%	2.49%	1.84%	0.95%	9.86%	467553
FATE2	5.38%	0.48%	0.67%	0.10%	6.63%	473259
FATE3	5.20%	1.37%	1.11%	1.40%	9.08%	469270
FATE4	5.22%	0.94%	0.99%	0.79%	7.95%	471249
FATE5	3.66%	2.74%	2.11%	2.46%	10.98%	468206

In the above scenarios, LRU, LFU, and FATE2 are the same algorithms as those used in the single-cache scenario. As mentioned previously, most on-path caching nodes are not used effectively, as shown by LRU's poor performance for nodes 2-4 (C2-C4) and LFU's declining performance over the same nodes. Even the algorithm FATE2, which was more effective in a single cache scenario, did not fare as well as LFU, over four on-path caching nodes. In order to more effectively use the nodes, the HASH evaluator was included in the FATE evaluation. Other algorithms may use an approximation of this effect: random or a probabilistic method, to determine if the content might be cached at each node, which may place content at zero, or multiple nodes.

Hashing avoids the zero/many placement problem associated with random/probabilistic cache placement, and allows other nodes to have higher or lower weight to keep the content. FATE5 uses the hash to determine cache placement using a modulus function (hash modulus 4 determines which node will cache the content), and is represented by $(0.2 + \text{LRU} \cdot 0.8) \cdot \text{LFU} \cdot \text{HASH}$. As shown by the hit rate per individual nodes, it is more efficient in the use of the on-path caching nodes, and gives over 11% benefit compared to LRU. FATE4 was an attempt to heavily bias caching via HASH, but allow other (popular content) to be cached, and is expressed by: $\text{MIN}((0.2 + \text{LRU} \cdot 0.8) \cdot \text{LFU}, 0.2 + 0.8 \cdot \text{HASH})$. With the result of having the second highest hit rate on cache node 1, but poor utilization of the other cache nodes. FATE3 is an attempt to use the FATE2 algorithm at node 1 (to maximize close caching), but FATE 5 algorithm at nodes 2, 3, and 4 (to allow more efficient on-path caching). This heterogeneous-caching algorithm allows for most popular content be cached closer to the client, while maximizing efficiency of the remaining on-path nodes.

Total Hit-rate represents the total of all on-path cache hits (as opposed from a server delivered content). Assuming a 1 ms delay, per request from the client to each node (i.e., client to node 1 is 1ms, client to server is 5ms), the total time required is represented by Total Response Delay(ms). Despite algorithm FATE5 delivering a better hit rate than LRU, LRU delivers the best (lowest) overall response time.

V. CONCLUSION

FATE provides a more flexible and rapid development environment for protocols and algorithms that evaluate and act upon Information. In this paper we concentrated on caching

using FATE. With FATE, new functionality can be added by adding new modules, and new evaluations can be created with new functions. The resulting algebraic function of the action, and through an extension the XML configuration file, help remove redundant or inaccurate implementations of the same networking algorithm. Our future application of FATE includes forwarding and security, and optimizing caching for low-power sensor networks. The FATE code is publicly available [11], and the code (NDNSim with FATE caching)/results for this paper are available at a different location [12].

ACKNOWLEDGMENTS

This work was supported in part by the Jack Baskin Chair of Computer Engineering at UC Santa Cruz.

REFERENCES

- [1] Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnSIM: NDN simulator for NS-3. Technical Report NDN-0005, NDN, October 2012.
- [2] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.
- [3] Raffaele Chiocchetti, Dario Rossi, and Giuseppe Rossini. ccnsim: An highly scalable ccn simulator. In *ICC*, pages 2309–2314. IEEE, 2013.
- [4] NS-3 Consortium. Network simulator ns3. Online; accessed 13-October-2015.
- [5] Ali Dabirmoghaddam, Maziar Mirzazad Barijough, and JJ Garcia-Luna-Aceves. Understanding optimal caching and opportunistic caching at the edge of information-centric networks. In *Proceedings of the 1st international conference on Information-centric networking*, pages 47–56. ACM, 2014.
- [6] Mohamed Hefeeda and Osama Saleh. Traffic modeling and proportional partial caching for peer-to-peer systems. *IEEE/ACM Transactions on networking*, 16(6):1447–1460, 2008.
- [7] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. Manet simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, 2005.
- [8] Nikolaos Laoutaris, Sofia Syntila, and Ioannis Stavrakakis. Meta algorithms for hierarchical web caches. In *Performance, Computing, and Communications, 2004 IEEE International Conference on*, pages 445–452. IEEE, 2004.
- [9] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.
- [10] Spyridon Mastorakis, Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnSIM 2.0: A new version of the NDN simulator for NS-3. Technical Report NDN-0028, NDN, January 2015.
- [11] James Mathewson. Fate - functional algebraic atomic evaluators. <http://github.com/jlmathew/Fate>. Online.
- [12] James Mathewson. Fate cache code and results for icnc2019. <https://drive.google.com/file/d/1FYRE7SCiEWbJfWC1Icy04sp1ulhLNW2g/view?usp=sharing>. Online.
- [13] Ioannis Psaras, Wei Koong Chai, and George Pavlou. Probabilistic in-network caching for information-centric networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, pages 55–60. ACM, 2012.
- [14] Ivan Stojmenovic. Simulations in wireless sensor and ad hoc networks: matching and advancing models, metrics, and solutions. *IEEE Communications Magazine*, 46(12):102–107, 2008.
- [15] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, Simutools '08*, pages 60:1–60:10. ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).